MASTER EN SCIENCES INFORMATIQUES

Étude méthodologique et réalisation d'un didacticiel extensible en support à un cours de théorie des graphes

Boucqueau, Dimitri

Award date:
2003

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Année académique 2002 – 2003

Etude méthodologique et réalisation d'un didacticiel extensible en support à un cours de théorie des graphes.

Dimitri Boucqueau

Mémoire présenté en vue de l'obtention du grade de
Licencié en Informatique

Résumé – Abstract

Dans ce travail, nous avons démontré l'utilisation d'une méthodologie avancée de développement d'un produit logiciel, cette méthodologie étant itérative et incrémentale.

Nous avons étudié, analysé et spécifié divers composants permettant de modéliser un graphe mathématique et divers algorithmes issus de la théorie des graphes. Ces composants, faisant partie d'un système ouvert et extensible, permettent de créer des applications utilisant graphes et algorithmes en ajoutant simplement ces composants dans les logiciels.

Nous avons également étudié, analysé et spécifié un composant faisant partie de la même bibliothèque qui permet de visualiser les graphes, ainsi que le fonctionnement des algorithmes sur ces graphes, et de les manipuler interactivement.

Nous avons réalisé ces composants en langage Delphi, et créé trois applications didactiques à titre de démonstration de la validité de notre système.

Mots-clés : théorie des graphes, algorithmes, méthodologie, processus unifié, visualisation de graphe, manipulation de graphe, bibliothèque de composants.

In this work, we demonstrated how we used an advanced methodology to develop a software product, this methodology being iterative and incremental.

We studied, analyzed and specified components allowing people to model a mathematical graph and some algorithms from the graph theory. These components, being part of an open and extensible system, allow developers to create applications using graphs and algorithms by simply plugging components in their software.

We also studied, analyzed and specified a component from the same library that allows users to visualize graphs, including how algorithms execute on these graphs, and manipulate them interactively.

We realized these components in the Delphi language, and created three didactic applications to demonstrate the correctness of our system.

Keywords: graph theory, algorithms, methodology, unified process, graph visualization, graph drawing, component library.

Avant-propos

Un travail de fin d'études universitaires n'est pas une entreprise aisée. Nous souhaitons remercier sincèrement les personnes suivantes, sans lesquelles nous n'aurions certainement jamais pu aller jusqu'au bout.

Monsieur Jean-Paul Leclercq, promoteur de ce mémoire : sans vos conseils avisés et votre expérience, ce mémoire ne serait tout simplement pas. Nos trop rares réunions m'ont fait prendre conscience d'un point si capital dans notre métier d'informaticien : le point de vue de l'utilisateur final doit toujours jouir de la plus grande importance. Un énorme merci pour vos conseils, votre franchise et votre bon sens. Je souhaite que ce travail vous soit utile pour vos cours, et j'espère que notre collaboration pourra survivre à la fin de mes études.

Madame Marie d'Udekem-Gevers : merci pour vos encouragements à aller jusqu'au bout et à ne pas baisser les bras. Et merci pour la « somme relative aux mémoires ». C'est un document qui a été bien utile dans la rédaction de ce travail !

Monsieur Pierre-Yves Schobbens : vos remarques lors du séminaire préparatoire à la présentation des mémoires ont été fort utiles, et j'en ai tenu compte pour améliorer ce travail. Merci pour cela !

Madame Nathalie Vandermeulen, ma compagne : sans toi, je n'aurais déjà jamais réussi à atteindre l'étape du mémoire. Sans tes sacrifices, je n'aurais jamais pu terminer ce mémoire non plus. Pour tout cela, je souhaite te remercier publiquement, bien que je l'aie déjà fait souvent en privé. Si tu souhaites faire tes études d'égyptologie, c'est à mon tour de m'occuper de tout à la maison !

Enfin, je souhaite remercier également mes amis, qui m'ont encouragé à aller jusqu'au bout : **Daniel Philippe, Olivier Hislaire, Joël Adam**.

Si j'ai oublié certaines personnes, qu'elles m'excusent en sachant que je leur serai toujours reconnaissant.

Table des matières

1

Introduction

Le projet en bref — Les objectifs de ce travail — La découpe du travail — Les conventions de notation — Glossaire

Le présent travail traite de la théorie des graphes, il est vrai, mais il est surtout basé sur une démarche rationnelle et méthodologique de développement d'un système logiciel. Nous nous proposons de démontrer comment nous avons réalisé un projet de réalisation de système logiciel, en expliquant la méthodologie utilisée.

1.1 Le projet en bref

Le projet sur lequel est basé ce travail consiste à analyser, spécifier et, accessoirement, réaliser un système de composants logiciels permettant de doter diverses applications de fonctionnalités tirées de la théorie des graphes. Tout cela doit se faire dans le paradigme « Orienté Objet ».

L'emphase est placée sur l'analyse et la conception : il est demandé de fournir ces résultats dans une optique d'abstraction par rapport à tout langage. Ainsi, en suivant les directives de conception, tout programmeur un tant soit peu chevronné dans son propre langage de programmation (que ce soit Delphi, Java, Powerbuilder, etc.) pourra réaliser ce système.

Aux fins de preuve du concept, nous avons également réalisé ce système de composants dans le langage Delphi, et créé quelques petites applications démontrant le potentiel du système.

1.2 Objectifs de ce travail

Ce travail va expliquer la démarche utilisée dans le cadre de l'analyse, de la conception et du développement de composants modélisant des éléments de la théorie des graphes.

Nous pensons que le résultat final a une certaine utilisabilité, mais nous estimons cependant que c'est la méthodologie et la démarche suivies qui ont le plus d'importance dans le cadre d'un mémoire de fin d'études universitaires.

C'est pourquoi la plupart des résultats concrets du projet se trouveront dans les annexes de ce travail, tandis que les explications sur la méthodologie, sur les problèmes rencontrés et leurs résolutions formeront le gros du texte principal.

1.2.1 La démarche

Nous avons opté pour une démarche rationnelle pour le projet. Cette démarche est basée sur le « Processus Unifié », mais elle a été évidemment adaptée pour s'accorder avec les caractéristiques particulières de ce projet : une équipe réduite, une portée limitée.

Le lecteur trouvera l'explication complète de la démarche suivie dans les pages de ce travail. Il pourra commencer par lire le chapitre 2, « La démarche », pour une explication de base. Ensuite, il trouvera des explications sur l'application concrète de cette démarche dans les chapitres suivants : les points clés de la démarche, les problèmes rencontrés et les solutions qui ont été apportées. Certaines parties jugées inutilement complexes ont été omises.

1.2.2 Les objectifs du projet

Le projet a les objectifs suivants :

- Modélisation abstraite des graphes : fournir l'analyse et la conception dans le paradigme « Orienté Objet » des divers éléments constitutifs de la théorie des graphes, ainsi que de certains algorithmes permettant de les rechercher. On demande également une abstraction complète par rapport à tout langage de programmation.

- Modélisation abstraite d'un visualisateur/manipulateur de graphe : fournir l'analyse et la conception dans le paradigme « Orienté Objet » d'un composant permettant de visualiser et manipuler des graphes. Ici aussi, on demande une abstraction complète par rapport à tout langage de programmation.

- Réalisation des divers composants : réaliser, en Delphi[1], les composants spécifiés et conçus lors des deux points précédents.

- Développement de petites applications : développer de petites applications démontrant le fonctionnement des composants réalisés lors du point précédent, ainsi que le potentiel du système.

1.3 Découpe du travail

Après cette introduction, nous proposons dans le chapitre 2 une explication détaillée de la démarche utilisée. Nous y indiquons ce qui a été utilisé, ce qui n'a pas été utilisé (et pourquoi). Nous tentons également d'expliquer pourquoi nous avons choisi cette démarche et les effets positifs qu'elle a eu sur le projet.

Le chapitre 3 traite de l'état de l'art dans le domaine traité. On y décrit ce qui existe, et on en effectue une analyse.

---

[1] Le langage Delphi (un langage orienté objet à syntaxe Pascal) a été choisi par souci de facilité et de rapidité de développement. Comme on l'a dit, la conception doit être suffisamment abstraite pour permettre de réaliser ces composants dans divers langages. Le choix de Delphi n'a dès lors aucun impact significatif sur le projet dans son ensemble.

Dans le chapitre 4, on trouvera les résultats de la phase de création du projet : comment le projet a été lancé, la proposition de base, l'étude du risque.

Au chapitre 5, on s'intéressera à la phase d'élaboration du projet : capture des besoins fonctionnels et non fonctionnels, analyse et conception architecturale.

Le chapitre 6 traite de la phase de construction : on y parle de l'analyse complète des besoins, de la conception du système et de la construction des composants.

C'est au chapitre 7 qu'on parlera de phase de transition : correction des anomalies rencontrées, livraison et utilisation du système fini.

Enfin, le chapitre 8 est une conclusion à ce travail. Nous y analysons ce qui a été réalisé par rapport à l'état de l'art ; nous y indiquons ce qui reste encore à faire, et ce que nous aurions pu faire autrement.

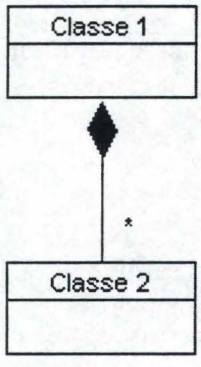
1.4 Conventions de notation

Nous proposons les conventions de notation suivantes dans ce travail :

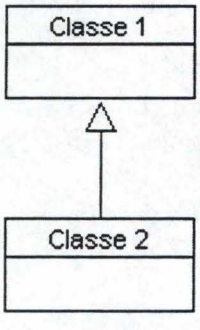
1.4.1 Diagrammes UML

Nous utilisons les diagrammes UML standards. En quelques mots, voici les significations des flèches et notations diverses.

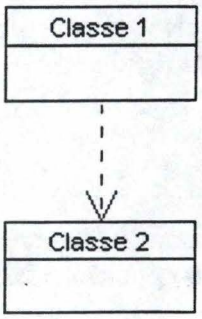
Un nom de classe en *italique* indique que cette classe est abstraite.



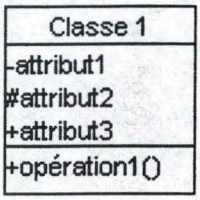
Le losange noirci indique une relation de « composition » : la classe 2 de notre exemple est un attribut n'ayant d'existence que pendant la durée de vie de la classe 1. L'étoile est une indication de « multiplicité » : elle signifie que la classe 1 peut contenir un nombre indéfini d'instances de la classe 2.



La flèche de notre exemple indique une relation de généralisation : la classe 1 est une version plus générale de la classe 2. *A contrario*, la classe 2 est une spécialisation de la classe 1, obtenue par héritage.



La flèche pointillée indique une relation de dépendance entre les classes. Généralement, on indique le type de dépendance grâce à une étiquette près de la flèche.



Une classe est représentée dans une boîte comprenant 3 parties : la première contient le nom de la classe, la seconde contient les attributs de la classe, et la dernière ses opérations. Certaines parties peuvent être vides, et ne sont alors pas montrées.

La visibilité des attributs et opérations est indiquée par des codes devant le nom de l'opération ou de l'attribut : « - » signifie « privé », « + » signifie « public », et « # » signifie « protégé ».

1.4.2 Notations dans le texte

Un nom d'attribut ou d'opération, ou encore un paramètre de méthode, est noté en *italique souligné*.

La notation *A→B* signifie l'attribut ou l'opération de l'instance de la classe A. La notation *A(...)→B* signifie que la méthode A renvoie un élément de type B.

Les types de base standards pour les attributs sont indiqués comme suit : « int » pour un nombre entier signé, « float » pour un nombre à virgule flottante, « boolean » pour une valeur booléenne, « String » pour une chaîne de caractères.

1.5 Glossaire

Algorithme : pour le présent projet, un algorithme est un ensemble d'opérations, généralement itératives, qui, appliquées à un ou plusieurs graphes, permettent de résoudre un problème donné, comme les problèmes de connectivité ou d'accessibilité, de cheminement (optimal ou particulier), de couverture, de coloration, etc. *Voir : Graphe.*

Arc : un arc du graphe G est un couple représentant la relation binaire entre deux sommets du graphe. On parle d'arc lorsque le graphe est orienté, et d'arête sinon. Graphiquement, on représente un arc par « un segment reliant les deux sommets, orienté au moyen d'une flèche. » [Leclercq JP, 1998]. *Voir : Graphe, Sommet, Arête, Orientation.*

Architecture : « Ensemble des décisions significatives concernant l'organisation d'un système logiciel, la sélection des éléments structurels dont est composé le système et de leurs interfaces, ainsi que leur comportement tel qu'il est spécifié dans les collaborations entre ces éléments, la composition de ces éléments structurels et comportementaux en sous-systèmes progressivement plus importants, et le style

architectural guidant cette organisation : ces éléments et leurs interfaces, leurs collaborations, et leur composition » [Jacobson I, et al, 2000]

Arête : une arête du graphe G est le couple d'arcs (x, y) et (y, x) de G représentant une relation symétrique. Graphiquement, on représente une arête comme un segment reliant les deux sommets, sans qu'il soit nécessaire d'indiquer les flèches (car superflues). On parle généralement d'arête lorsque le graphe n'est pas orienté. *Voir : Graphe, Arc, Sommet, Orientation.*

Besoin : un besoin est « une condition ou une capacité à laquelle doit se conformer un système » [Jacobson I, et al, 2000]. On distingue les besoins fonctionnels et les besoins non fonctionnels. Les besoins fonctionnels « spécifient une action qu'un système doit être capable d'effectuer, sans considérer aucune contrainte physique » [Jacobson I, et al, 2000]. Les besoins non fonctionnels spécifient « des propriétés du système, telles que les contraintes liées à l'environnement et à l'implémentation, et les exigences en matière de performances, de dépendances de plate-forme, de facilité de maintenance, d'extensibilité et de fiabilité. » [Jacobson I, et al, 2000]

Cas d'utilisation : un cas d'utilisation est le résultat d'une « technique d'élaboration des besoins fonctionnels, selon le point de vue d'une catégorie d'utilisateurs (...) Ils décrivent, sous la forme d'actions et de réactions, le comportement d'un système du point de vue d'un utilisateur » [Muller PA, et al, 2000]. *Voir : Besoin.*

Chemin (circuit) : un chemin est « une suite de n arcs (x,y), (y,z), (z,t)... tels que l'extrémité du $k^{ème}$ arc soit l'origine du $(k+1)^{ème}$ lorsque la longueur du chemin (çàd le nombre d'arcs dans le chemin) est supérieure à 1 » [Leclercq JP, 1998]. Un circuit est « un chemin dont l'origine du premier arc coïncide avec l'extrémité du dernier » [Leclercq JP, 1998]. Remarquons qu'on parle de chemins et circuits dans le cas de graphes orientés ; pour les graphes non orientés, on parle respectivement de chaînes et de cycles. *Voir : Arc, Orientation.*

Etiquette : l'étiquette d'un sommet, d'un arc ou d'une arête du graphe G, est un texte que l'on associe à cet élément, et pouvant représenter, par exemple, le nom du sommet, l'opération que représente un arc, etc. Le contenu de cette étiquette est laissé libre à l'utilisateur et n'a généralement pas d'impact sur les algorithmes. *Voir : Sommet, Arc, Arête.*

Evénement : un événement est un lien entre quelque chose qui se passe dans le système (comme une action de l'utilisateur) et le code qui y répond. Ce code est un « gestionnaire d'événement » et est presque toujours écrit par le développeur d'application. Les événements permettent aux développeurs de personnaliser le comportement de composants dans devoir changer leurs classes. Généralement, les événements sont des objets eux-mêmes, et fournissent des données permettant de traiter l'événement précis.

Graphe : « un graphe G est la représentation (visualisée au moyen d'un graphique) d'une relation binaire sur un ensemble X, c'est-à-dire une partie du produit cartésien de X par lui-même, ou encore un ensemble U de couples de X. Les éléments de X sont représentés dans le plan par des points appelés sommets et la relation entre deux sommets par un arc » [Leclercq JP, 1998] *Voir : Sommet, Arc.*

Informations d'habillage : il s'agit de l'ensemble des informations permettant de décrire la façon dont un graphe G donné doit être affiché. On y retrouve : l'emplacement des sommets et des diverses étiquettes, la façon dont on représente un sommet, la taille des sommets et des arcs, la fonte des étiquettes, les couleurs à employer, etc. On parle parfois d'informations de « décoration ». *Voir : Arc, Arête, Sommet, Etiquette.*

Modèle : un modèle est « une construction descriptive à partir d'éléments de modélisation » [Muller PA, et al, 2000]. C'est « une abstraction d'un système, décrivant le système modélisé d'un certain point de vue et à un certain niveau d'abstraction » [Jacobson I, et al, 2000].

Orientation : l'orientation d'un graphe G est un caractère exprimant le fait que la relation qu'il représente est symétrique (le graphe n'est alors pas orienté) ou non (le graphe est alors orienté). *Voir : Graphe.*

Poids : le poids d'un sommet ou d'un arc ou arête du graphe G représente l'information de pondération de cet élément du graphe. Selon les cas, le poids est une valeur entière ou à virgule flottante, et peut ou non être négative, positive ou nulle. *Voir : Graphe, Sommet, Arc, Arête, Pondération.*

Pondération : la pondération d'un graphe G est un caractère indiquant si ses sommets (rarement) et/ou ses arcs ou arêtes (le plus souvent) possèdent en plus une information de valuation, appelée poids. *Voir : Graphe, Sommet, Arc, Arête*

Représentation interne : une représentation interne est la manière dont un graphe G donné est décrit informatiquement selon une structure de données particulière. Il existe plusieurs types courants de représentations internes : la matrice d'adjacence, les listes de pointeurs, etc. *Voir : Graphe.*

Risque : dans le cadre de ce projet, nous définissons le risque sous sa forme de « risque concernant le développement informatique », à savoir « une menace potentielle ou un événement pouvant empêcher d'atteindre les objectifs définis ; il peut affecter les délais, les coûts, la qualité ou les bénéfices » [Buttrick R., 2000]

Sommet : un sommet du graphe G est un élément de l'ensemble X dont le graphe représente une relation binaire sur lui-même. Graphiquement, on représente généralement un sommet par un point. *Voir : Graphe.*

Théorie des graphes : la théorie des graphes est la discipline traitant des graphes mathématiques, des algorithmes qu'on peut appliquer à ces graphes, et de tous les problèmes liés aux graphes mathématiques (représentation, complexité, etc.) *Voir : Graphe, Algorithme.*

V/M (visualisateur/manipulateur) : un visualisateur/manipulateur est un composant informatique du système que le projet doit développer, et qui permet à l'utilisateur final de voir une représentation graphique d'un graphe G donné et de manipuler ce même graphe (en déplaçant ses éléments, et en créant, modifiant et supprimant des éléments du graphe). Dans le reste du texte, nous utiliserons souvent l'abréviation « V/M » pour éviter la longue formulation « visualisateur/manipulateur ». *Voir : Graphe, Informations d'habillage.*

2

La méthodologie

Pourquoi une méthodologie – La méthodologie choisie – Vue d'ensemble de la méthode – Notre adaptation de la méthodologie

Ce chapitre va décrire la méthodologie générale qui a été utilisée pour mener le projet global à son terme. Les divers points de cette méthodologie seront explicités en détail dans les chapitres suivants, mais il nous semble qu'une vue d'ensemble est utile dès maintenant.

2.1 Pourquoi une méthodologie ?

Une méthodologie de développement logiciel est maintenant indispensable, dans le but de produire les applications et systèmes voulus dans les contraintes de budget, de délai et de coût qui leurs sont associées. Le temps de l'amateurisme, tacitement acceptée pendant que la discipline d'ingénierie logicielle en était encore à ses balbutiements, est révolu. Le monde moderne veut produire des systèmes en appliquant un processus répétable, contrôlé et sûr.

2.1.1 Avantages d'une méthodologie

L'utilisation d'un processus ou d'une méthodologie standard procure de nombreux avantages :

- Chaque personne (les êtres humains qui « sont les éléments moteurs de tout projet logiciel » [Jacobson I., et al, 2000], comme les architectes, développeurs, testeurs, utilisateurs, etc.) connaît sa place et son rôle dans le développement du produit.

- Les développeurs ont une meilleure compréhension de l'activité des autres développeurs, que ce soit dans le même projet ou dans un autre projet, à des stades divers du cycle de vie.

- L'utilisation d'un langage de modélisation permet aux dirigeants de comprendre ce que font les développeurs.

- Les personnes peuvent passer d'un projet à l'autre sans devoir apprendre à chaque fois un nouveau processus.

- La formation au processus peut être standardisée.

- Et surtout, « le déroulement du développement logiciel est reproductible », avec tous les impacts positifs en matière d'expérience pour les activités annexes (quoique importantes) de planification et de budgétisation.

2.1.2 Outils

Un autre avantage de l'utilisation d'une méthodologie est l'utilisation maintenant quasi-systématique d'outils de développement.

Comme le disent Grady, Booch et Jacobson dans [Jacobson I., et al, 2000], « les outils influent sur le processus » et « le processus conditionne les outils ». Il y a donc un équilibre à trouver pour que ces outils, légitimés par le processus, soient suffisamment simples d'utilisation tout en étant utiles.

Les outils permettent[2] de maintenir la cohérence pendant tout le cycle de vie ; ils autorisent également l'automatisation de certaines activités, améliorant ainsi qualité et productivité.

Pour ce projet, nous avons employé un outil de modélisation appelé « Visual Paradigm for UML », en version « Community ». [Visual Paradigm, 2003]

2.2 La méthodologie choisie

2.2.1 Quelle méthodologie pour ce projet ?

La méthodologie que nous avons décidé de suivre pour ce projet est une adaptation du « Processus Unifié ». Adaptation dans le sens où le processus complet est trop lourd pour un projet de cette taille : un seul développeur, qui cumule les diverses activités, et un projet de taille modeste.

Ainsi, certains points du processus ont été purement et simplement éliminés lorsqu'il nous semblait que leur valeur ajoutée était nulle ; d'autres ont été simplifiés pour ne pas pénaliser inutilement le travail.

2.2.2 Pourquoi nous avons choisi cette méthodologie ?

Le choix du Processus Unifié comme méthodologie de base s'est fait assez naturellement :

1. Le processus est facilement adaptable aux circonstances (on peut le voir comme « un *framework* de processus générique pouvant être adapté à une large classe de systèmes logiciels, à différents domaines d'application, à différents types d'entreprises, à différents niveaux de compétence et à différentes tailles de projets » [Jacobson I., et al, 2000]) ;

2. Il est spécialisé dans les développements orientés objet ;

[2] Ou, en tout cas, devraient permettre…

3. Il est itératif et incrémental, ce qui est intéressant dans le cadre d'un travail « de soir » pouvant être interrompu par moments pour diverses raisons familiales ou professionnelles ;

4. Il est issu d'une longue évolution de travaux, remontant jusqu'aux années 1980, ce qui est à notre sens une garantie de qualité (point subjectif et donc discutable s'il en est...) ;

5. Nous avons un intérêt particulier à pratiquer cette méthodologie pour nos activités professionnelles.

2.3 Vue d'ensemble de la méthode

2.3.1 Les idées de base

Le processus unifié est **à base de composants**, c'est-à-dire que le système que l'on va fabriquer est fait de composants logiciels reliés les uns aux autres par des interfaces clairement définies.

Il est **piloté par les cas d'utilisation**. Il faut bien comprendre les désirs et les besoins des futurs utilisateurs pour construire le système qui leur rendra réellement service ; les cas d'utilisation, réalisés grâce au langage de modélisation UML (*Unified Modeling Language*), remplacent les classiques spécifications fonctionnelles. Ils permettent de saisir les besoins fonctionnels et non fonctionnels **de chaque utilisateur**, dans leur propre langue, ce qui oblige les analystes à « réfléchir en termes d'avantages pour les utilisateurs » [Jacobson I., et al, 2000]. De plus, outre la spécification des besoins du système, ils guident la conception, l'implémentation et les tests, c'est-à-dire le processus de développement.

Il est **centré sur l'architecture**. En fait, l'architecture logicielle (qui représente la forme du produit logiciel en cours de fabrication) « émerge des besoins » [Jacobson I., et al, 2000] reflétés par les cas d'utilisation (qui représentent la fonction du produit), mais « subit néanmoins l'influence d'autres facteurs, tels la plate-forme d'exécution du produit, ou l'utilisation de *framework* » [Jacobson I., et al, 2000]. Elle « propose donc une vue d'ensemble de la conception en faisant ressortir les caractéristiques essentielles en laissant de côté les détails secondaires ». Le but est d'arriver à « couler le système dans une forme, qui doit être conçue de façon à permettre l'évolution du système, non seulement dans le cadre de son développement initial, mais dans les années et les générations à venir » [Jacobson I., et al, 2000].

Il est **itératif et incrémental**. Il permet ainsi de découper le travail global en plusieurs parties : chacune représente une itération (un enchaînement d'activités) donnant lieu à un incrément[3] (un stade de développement du produit). Evidemment, tout cela doit être soigneusement sélectionné et planifié. Le choix de ce qui doit être implémenté au cours d'une itération est important : il faut prendre en compte un certain nombre de

[3] Notons que « un incrément ne constitue pas nécessairement un additif : dans les premières phases du cycle de vie, (...) il n'est pas rare de remplacer une conception superficielle par une autre plus détaillée ou plus complexe » [Jacobson I., et al, 2000].

cas d'utilisation qui améliorent l'utilisabilité du produit, et traiter en priorité les risques majeurs. L'utilisation d'un processus itératif est avantageuse :

- Elle permet de limiter les risques de retard dans le développement du produit en identifiant et traitant les risques dès les premiers stades ;
- Elle accélère le rythme : les développeurs travaillent sur des objectifs clairs à court terme, plutôt que sur la base d'un planning à long terme ;
- Elle permet, surtout, de gérer le changement : les besoins ne peuvent être tous définis à l'avance, et certains se dégagent peu à peu des itérations successives.

2.3.2 Le cycle de vie du processus

Un cycle du Processus Unifié fournit une nouvelle version d'un système aux clients. Plusieurs cycles peuvent se suivre, faisant ainsi évoluer le système dans le temps.

Un cycle est constitué de 4 phases : création, élaboration, construction et transition. Chacune de ces phases se subdivise à son tour en plusieurs itérations.

Chaque itération va généralement mettre en œuvre cinq enchaînements d'activités : besoins, analyse, conception, implémentation, tests. L'importance relative de chaque enchaînement d'activité varie selon l'itération et la phase du cycle de vie : par exemple, peu ou pas de tests sont effectués pendant les itérations de la phase de création, tandis qu'ils prennent une grande importance vers les dernières itérations de la phase de construction.

La figure suivante, extraite de [Jacobson I., et al, 2000] résume utilement les points que nous venons de voir :

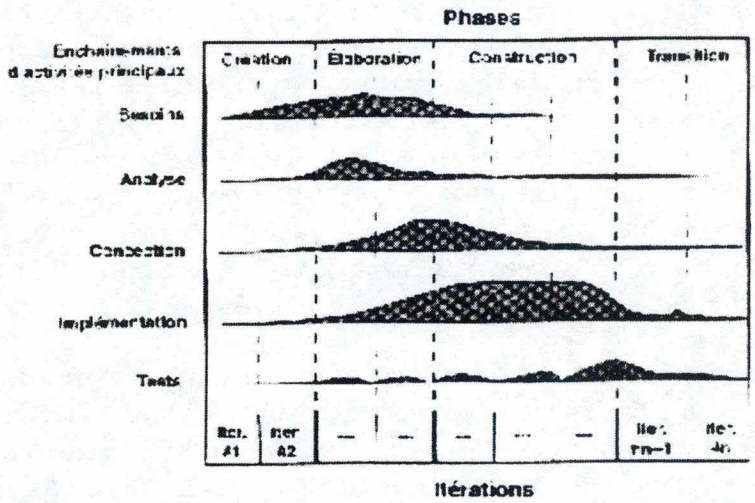


Figure 2-1 : importance des activités selon les phases du Processus Unifié.

Nous nous trouvons donc loin du cycle de vie de développement logiciel habituel (le fameux cycle de vie « en V »), avec son processus séquentiel et ses phases bien distinctes (expression des besoins, spécifications, conception générale, conception

détaillée, réalisation, tests unitaires, tests d'intégration, validation, maintenance) où les problèmes n'apparaissent que tard, et ne travaillant que dans le long terme hypothétique (l'effet « tunnel »).

2.3.3 La phase de création en bref

La phase de création va transformer une idée de départ (que nous appelons la « proposition ») en une vision du système qu'il faut développer. Elle doit également fournir une étude de rentabilité du produit, ce qui permet d'arrêter le projet avant que de lourds investissements trop hypothétiques ne soient lancés ou si le projet ne satisfait pas à la stratégie de l'entreprise.

On souhaite obtenir, à la fin de cette phase, un modèle simplifié des cas d'utilisation, qui contiendra les principaux cas d'utilisation du système, ce qui permettra de répondre à la question : « que va faire, en substance, le système pour chacun de ses principaux utilisateurs ? ».

On souhaite également obtenir une ébauche de l'architecture du futur système, révélant les principaux sous-systèmes. Encore provisoire à ce stade, cette architecture sera conçue en détail lors des phases suivantes.

Enfin, on demande également d'identifier les risques majeurs du projet.

Tout cela permet alors de planifier la phase d'élaboration qui va suivre, et fournir une estimation approximative du projet global.

En résumé, la phase de création va principalement identifier les besoins de base du système, et étudier la rentabilité du projet. Parfois, on effectue de légères analyses et on réalise un prototype très léger, mais ces dernières activités sont plus accessoires.

2.3.4 La phase d'élaboration en bref

Le but de cette phase est de définir la plupart des cas d'utilisation du système, et de concevoir l'architecture du système.

Il faut bien comprendre que certains cas d'utilisation n'apparaîtront que plus tard (dans la phase de construction), car les utilisateurs peuvent ne pas avoir pensé à tout lors de la capture des besoins, ou encore parce que des changements peuvent survenir.

L'architecture qui sera conçue dans cette phase devra être suffisamment stable pour que le chef de projet, en l'associant aux cas d'utilisation et à l'étude des risques, puisse décider et planifier la poursuite des activités. Cette architecture de référence est l'ossature du système, que l'on étoffera dans la phase suivante ; elle est donc très importante, car c'est elle qui conditionne à la fois le reste du cycle de vie du projet, mais aussi les cycles suivants selon son évolutivité et son extensibilité.

En résumé, cette phase est celle de la capture des besoins, de leur analyse et d'une bonne partie de la conception du système (l'architecture de référence).

2.3.5 La phase de construction en bref

La phase de construction, la plus gourmande généralement en terme de ressources, est celle où le produit fini est construit : on transforme l'architecture de base en système complet en développant composants et sous-systèmes et en les intégrant.

Certaines modifications de l'architecture peuvent être apportées, mais n'oublions pas que cette architecture de référence doit être considérée comme stable à la fin de la phase d'élaboration ; ces changements doivent donc être mineurs.

Elle est découpée en plusieurs itérations ; chaque itération va fournir une version du système réalisant de nouveaux cas d'utilisation par rapport à la version précédente.

En résumé, cette phase est principalement consacrée à l'implémentation et aux tests du système ; quelques changements dans les besoins peuvent encore apparaître, ce qui implique des activités de capture des besoins, d'analyse et de conception, mais elles sont ici accessoires par rapport à l'activité de construction.

2.3.6 La phase de transition en bref

La phase de transition voit le produit testé par quelques utilisateurs expérimentés ; le but est de détecter et de corriger défauts et anomalies. On peut également y proposer des améliorations, qui seront implémentées dans ce cycle du processus ou dans les éventuels cycles suivants.

C'est aussi la phase de formation des utilisateurs, de fabrication (pour les produits non uniques), et de maintenance.

En résumé, cette phase comprend principalement des tests en exploitation réelle, et quelques activités de construction (les corrections d'anomalies détectées).

2.4 Notre adaptation de la méthodologie

La méthodologie complète du Processus Unifié est résumée dans l'annexe 1, « Les activités du Processus Unifié ».

Cependant, vu la taille du projet, nous avons apporté quelques adaptations à cette méthodologie, généralement en la simplifiant et en l'assouplissant. Nous allons expliciter maintenant les adaptations les plus importantes et les justifier.

L'application effective de la méthodologie adaptée sera expliquée au début de chacun des chapitres 4 à 7.

2.4.1 Un seul travailleur réel

La taille du projet nous autorise à n'avoir qu'une seule personne physique réelle qui effectuera chacun des rôles des divers travailleurs de la méthodologie.

2.4.2 Peu de prototypage de l'interface utilisateur

Le projet global étant plus qu'à moitié consacré à la spécification et la création de composants non visuels, et suite à des contraintes de temps et de distance, les activités de prototypage de l'interface utilisateur, que l'on retrouve dans l'enchaînement d'activités d'expression des besoins, n'ont été que peu effectuées.

2.4.3 Pas d'étude de rentabilité

Comme le projet n'a aucune vocation commerciale ou économique (puisque destiné à démontrer un savoir-faire dans le cadre d'un mémoire de fin d'études), il n'y a aucun besoin d'effectuer d'étude de rentabilité. Le projet est considéré comme acceptable dès le moment où le thème du travail de fin d'année est défini.

2.4.4 Peu de formalisme dans les tests

Le projet est de faible envergure et ne possède qu'un seul travailleur physique. Nous avons donc opté pour un très faible formalisme des tests. Nous décrirons les tests minimaux à effectuer avec succès pour que les composants soient considérés comme acceptés, mais sans pousser les détails des entrées et sorties attendues, ni des marches à suivre pour réaliser ces tests.

3

L'état de l'art

Les systèmes analysés – Analyse comparative des systèmes – Points clés de l'analyse

Tout travail se doit de commencer par une étude de l'état de l'art dans le domaine traité.

3.1 Les systèmes analysés

Les systèmes suivants ont été découverts lors de nos recherches. Nous donnons quelques mots d'explications sur chacun.

3.1.1 GRIN

GRIN [GRIN, 2003] est le système le plus proche de notre vision didactique : une application permet de construire interactivement des graphes mathématiques, d'appliquer des algorithmes sur ces graphes, de les sauver et lire à partir du disque dur.

La visualisation est claire, aérée et simple. L'utilisateur a accès à la fois à la visualisation et à la matrice d'adjacence du graphe.

Par contre, le produit nous semble assez lourd à utiliser. Les graphes sont limités à 250 sommets (ce qui est quand même largement suffisant pour un but didactique). Il n'y a pas de visualisation du fonctionnement d'un algorithme sur le graphe, mais uniquement des données textuelles. Enfin, il s'agit d'une application monolithique, et pas de composants réutilisables ; dans notre sens, il n'y a donc pas d'extension possible du produit tel quel.

3.1.2 aiSee

aiSee [aiSee, 2003] est un produit commercial, fort évolué au niveau de l'affichage des graphes. Cependant, il apparaît qu'il est impossible de construire interactivement des graphes à partir de ce produit : il se base sur une description en GDL (*Graph Description Language*, un standard de description textuelle de graphes) pour générer l'affichage. Il dispose de nombreux algorithmes de « réaffichage » prenant un graphe en entrée et générant une nouvelle organisation des sommets et arcs plus lisible.

Le produit possède de nombreux algorithmes de réaffichage puissants, et n'a pas de limite concernant les éléments qu'il traite. Il offre une excellent lisibilité, notamment grâce aux sous-graphes qu'on peut « replier » et « déplier » à volonté. Il donne une réelle impression de puissance.

Par contre, il n'y a aucune possibilité de créer un graphe interactivement (il faut passer par une description en GDL). Le produit apparaît également assez lourd à utiliser (peut-être à cause de sa puissance et de ses possibilités).

3.1.3 JDSL

JDSL [JDSL, 2003] adopte notre approche d'un ensemble de composants objets (ici écrits en Java) permettant de modéliser des graphes et de les traiter.

Dans l'état actuel de son développement (en version académique), il nous semble assez similaire dans le traitement des graphes. Cependant, il n'offre pas de visualisation et de manipulation de graphes ; tout doit être programmé dans l'application se basant sur ces librairies.

Il offre 4 types d'algorithmes : DFS, Dijkstra, tri topologique (Knuth) et arbre de recouvrement minimum (Prim-Jarnik).

3.1.4 Directed Graph Editor

Ce petit produit [DGEA, 2003] est en fait une *applet* écrite en Java permettant de dessiner interactivement des graphes. Assez simple à utiliser, elle semble par contre assez pauvre en fonctionnalités. Il n'y a aucun algorithme à appliquer. En fait, ce produit ressemble plus à une petite démonstration de Java qu'à un véritable produit de modélisation de la théorie des graphes.

3.1.5 ILOG JViews

ILOG Jviews [Jviews, 2003] est, comme aiSee (voir 3.1.2) un produit commercial, très puissant au niveau de l'affichage, mais n'offrant pas de possibilité de manipulation ou de garnissage interactif. Il propose lui aussi de nombreux algorithmes de repositionnement ou réaffichage des graphes qui lui sont fournis.

3.1.6 VGJ

« Visualizing Graphs with Java » [VGJ, 2003] est un système permettant de créer interactivement des graphes via une *applet* Java, et de leur appliquer des algorithmes de réaffichage. Il permet également de sauver les informations du graphe dans un fichier sous un format propriétaire appelé GML, et de lire des graphes à partir de tels fichiers. Cependant, il n'offre aucune possibilité de lui appliquer des algorithmes de traitement de graphes, ni d'étendre le système.

3.1.7 OpenJGraph

Ce produit [OpenJGraph, 2002], de type *open source*, ressemble très fort à notre projet : une librairie d'objets (ici en Java) permettant de créer et manipuler des graphes. Toujours en développement, il permet (ou permettra dans sa version finale) de créer des graphes par programmation ou interactivement grâce à un outil prévu à cet effet, d'appliquer des algorithmes sur ces graphes, etc.

3.2 Analyse comparative des systèmes

Nous proposons ici de réaliser cette étude en analysant les divers travaux découverts lors de nos recherches selon les points suivants :

- Graphe : peut-on modéliser des graphes ?
- Représentations internes : peut-on traiter plusieurs types de représentations internes pour un graphe ?
- Algorithmes : peut-on appliquer des algorithmes aux graphes créés ?
- Visualisation : peut-on visualiser des graphes ?
- Manipulation : peut-on manipuler des graphes de façon interactive ?
- Extensibilité : peut-on étendre les fonctionnalités du système ?

Les points examinés n'ont aucune prétention à l'exhaustivité, et ne représentent qu'un point de comparaison sur les besoins exprimés dans le projet actuel.

Le tableau suivant contient une analyse comparative des systèmes découverts.

Produit	GRIN	aiSee	JDSL	DGF	Inews	VGJ	OpenJGraph
Graphe	+	--	+	+	--	+	+
Repr. Internes	--	?	=	--	?	--	=
Algorithmes	+	?	+	--	?	--	+
Visualisation	++	++	--	=	++	+	+
Manipulation	=	--	--	+	--	+	+
Extensibilité	?	?	++	?	?	--	+

Tableau 3-1 : analyse comparative des produits découverts. « ++ » signifie que le produit est excellent dans ce domaine, « + » signifie un bon produit, « = » signifie un produit moyen, « - » signifie que le produit est faible, et « -- » signifie que le produit est mauvais (ou n'offre pas de possibilités dans ce domaine). « ? » indique qu'on n'a pas pu tester le produit pour ce domaine.

3.3 Points clés de l'analyse

3.3.1 Utilisation de standards

Il semble que pas mal d'applications analysées ne s'occupent pas de la création des graphes, mais uniquement de leur affichage. Le grand standard qui se dégage de notre analyse est le langage de description GDL. Il serait intéressant de permettre son utilisation dans notre produit.

3.3.2 Esthétisme de l'affichage

On peut dégager les produits en deux parties : ceux qui s'occupent uniquement d'afficher des graphes qu'on leur fournit en entrée selon divers canons esthétiques (représentation hiérarchique, orthogonale, etc.) et ceux qui permettent de créer et garnir les graphes interactivement. Ces derniers produits laissent l'utilisateur arranger lui-même son graphe.

Malgré le fait que notre produit tombe totalement dans la seconde catégorie, il semble utile de le prévoir de façon à pouvoir ajouter des algorithmes de « réaffichage » utilisés à la demande de l'utilisateur.

3.3.3 Représentations des éléments du graphe

Les éléments de base des graphes, c'est-à-dire les sommets et les arcs, peuvent avoir de très nombreuses représentations différentes. Outre les « simples » représentations purement géométriques (un sommet sous la forme d'un cercle, triangle, carré, losange, etc.), on trouve des applications où un sommet représente une machine dans un réseau informatique, ou une classe d'un diagramme UML, etc.

Cela implique que les éléments de base doivent pouvoir être étendus avec de nouvelles données, parfois très complexes.

4

La phase de création

La démarche méthodologique – La proposition – La liste des caractéristiques souhaitées – L'étude du risque – Le modèle du domaine

Ce chapitre traite de la phase de création. On y découvre la proposition à la base du projet, et les premiers résultats importants dans le cycle de vie du projet.

Nous expliquons d'abord la démarche méthodologique, puis nous exposons les résultats concrets pour ce projet.

4.1 La démarche méthodologique

La phase de création pour le projet en cours a nécessité une seule itération. Nous allons ici décrire la démarche méthodologique suivie.

4.1.1 Rédiger la proposition

Tout projet commence par une idée, une proposition, qu'il est important de rédiger.

Pour un projet d'envergure ou ayant une portée économique, il importe de rechercher une personne ayant avantage à la réalisation du projet concerné par la proposition ; cette personne est nommée « sponsor du projet ». Dans notre cas, on peut estimer que le sponsor du projet était M. Leclercq, promoteur du mémoire.

4.1.2 Elargir la vision du système

Une fois que la proposition représente un bénéfice potentiel pour le sponsor du projet, elle est étudiée plus en détail, et on en arrive à exprimer un certain nombre de besoins, d'exigences sur le projet qui en découle.

On les rédige ici sous la forme d'une liste des caractéristiques souhaitées.

4.1.3 Effectuer l'étude du risque pour le projet

Tout projet implique des risques. Il est donc important de les identifier dès le début et de définir les actions à prendre pour les réduire autant que possible.

4.1.4 Elaborer un modèle du domaine

Cette étape permet « aux développeurs d'avoir une réelle compréhension du contexte dans lequel se déploiera le système. » [Jacobson I., et al, 2000]. On distingue deux approches : la modélisation du domaine, et la modélisation du métier.

« Un modèle du domaine décrit, en les reliant les uns aux autres, les concepts essentiels du contexte sous forme d'objets du domaine. Une fois identifiés et nommés, ces objets permettent d'arrêter un glossaire qui favorisera la communication entre les diverses personnes travaillant sur le système [...] et faciliteront, ensuite, l'identification de certaines des classes au cours de l'analyse et de la conception du système ». [Jacobson I., et al, 2000]

Un modèle du métier est « un sur-ensemble du modèle du domaine. » [Jacobson I., et al, 2000] Il spécifie « les processus métier qui devront être pris en charge par le système. Outre l'identification des objets [...] impliqués dans l'activité professionnelle, la modélisation du métier établit également les compétences requises par chaque processus : les travailleurs, leurs responsabilités et les tâches qu'ils effectueront. » [Jacobson I., et al, 2000].

4.2 La proposition

L'idée de base de ce projet est née suite à une demande de propositions de sujets de mémoire. À cette époque, nous avions reçu notre cours de « Théorie des Graphes ». Nous avions alors trouvé désolant de ne pouvoir profiter d'une application informatique permettant d'illustrer visuellement le cours : construction interactive d'un graphe, visualisation et manipulation de ce graphe, et surtout visualisation du fonctionnement des divers algorithmes qui nous étaient présentés.

En effet, pour illustrer le fonctionnement d'un algorithme, nous étions forcés d'utiliser le tableau : tâche lente et fastidieuse, ne nous permettant généralement d'utiliser que de petits graphes simplistes de quelques sommets par faute de temps.

Notre idée fut donc de proposer de réaliser un ensemble de composants que l'on pourrait employer dans la construction de telles applications à caractère didactique. Afin d'étoffer le sujet, nous nous sommes imposé la conception et la spécification complète d'un système permettant de modéliser, dans le paradigme « Orienté Objet », la théorie des graphes (graphes, chemins, algorithmes) en faisant abstraction de tout langage de programmation. Il était également demandé de fournir un composant permettant de visualiser et manipuler les graphes, et d'afficher le fonctionnement d'algorithmes sur les graphes.

La réalisation d'applications est laissée à d'autres travaux qui viendront compléter, étendre ou utiliser les résultats de ce projet.

4.3 La liste des caractéristiques souhaitées

Cette étape permet de dresser une liste des suggestions, à considérer comme un ensemble d'exigences potentielles à implémenter. Elle sert principalement à planifier le début du travail.

Une réflexion personnelle et un entretien avec le sponsor du projet ont permis de sortir une liste de ces besoins potentiels. Cette liste était même assez détaillée pour certains points (comme la représentation à l'écran des sommets, arcs et étiquettes, par exemple). Le lecteur trouvera cette liste à l'annexe 2.

Nous insistons sur le fait que cette liste n'est pas un cahier des charges, et qu'elle ne représente que des exigences potentielles qui seront approuvées ou pas lors de la recherche des besoins fonctionnels et non fonctionnels grâce aux cas d'utilisation.

4.4 L'étude du risque

Une rapide étude du risque a été réalisée pour ce projet.

Le risque technologique est très faible, voire nul : le langage de programmation utilisé nous est bien connu, ainsi que les concepts du paradigme « orienté objet ».

Le risque de réaliser un produit inadapté est également assez faible : nous avons une idée très précise de ce que nous souhaitons obtenir comme résultat, et de plus le produit sera pensé et réalisé avec l'extensibilité à l'esprit ce qui permettra de rajouter *a posteriori* les éléments oubliés.

Le risque lié à « l'équipe de développement » est moyen : une seule personne pour un projet non trivial et nécessitant plus que quelques jours de développement, de plus cette personne n'est pas affectée à plein temps sur ce seul projet. Cependant, les délais sont suffisamment larges pour réduire l'importance de ce risque ; on suggère des petites itérations bien déterminées et des jalons officiels afin de garder le rythme du développement et réduire ce risque.

Le risque technique est également faible : l'idée est surtout de démontrer un savoir-faire et de fournir des spécifications pour un système. Même la réalisation effective de ce système n'est pas subordonnée à des exigences très fortes en matière de robustesse du système ou d'optimisation du temps d'exécution.

En résumé, le projet est peu risqué ; la plus importante des actions à entreprendre consiste à s'assurer un rythme de développement suffisamment suivi pour ne pas dépasser les délais.

4.5 Estimation du temps de développement

Il est difficile de faire une estimation réaliste du temps de développement à ce stade, étant donné que les cas d'utilisation ne sont pas encore définis. Cependant, en nous basant sur notre expérience, on peut avancer le chiffre de deux mois/homme afin d'avoir un travail soigné. Ce chiffre ne vaut que pour les activités de développement pure (analyse, conception, codage, tests) et ne tient pas compte du temps nécessaire à rédiger le texte de ce mémoire.

4.6 Le modèle du domaine

Dans notre projet, un modèle du domaine est largement suffisant. Il est établi sous la forme du diagramme de classes UML suivant (voir figure 4-1).

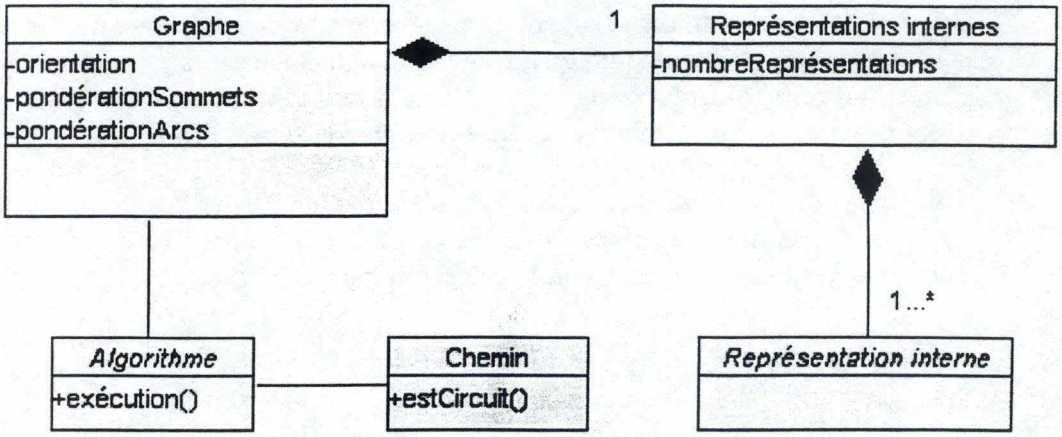


Figure 4-1 : Modèle simplifié du domaine pour le projet.

Nous voyons tout d'abord que l'élément principal du domaine est, sans surprise, le graphe. Celui-ci possède quelques attributs de base, comme son caractère orienté ou non ou la possible pondération de ses sommets et/ou arcs. Il possède également, par composition, un certain nombre de représentations internes (au moins une à tout moment).

Le second élément important est représenté par les algorithmes, qui s'appliquent aux graphes et peuvent fournir des résultats entres autres sous la forme de chemins (ou circuits). Notons ici que dès le départ, il est prévu que l'on conçoive le système avec une classe « Algorithme » de base abstraite, et que les véritables algorithmes seront créés sous la forme de classes dérivées de cette super-classe abstraite.

De même, les représentations internes sont représentées par une classe abstraite de laquelle seront créées les véritables représentations internes via héritage. Ce modèle est suffisamment souple pour permettre un passage en revue de toutes les représentations internes lors, par exemple, d'une opération de garnissage ; en effet, dans ce cas, toutes les représentations internes doivent être mises à jour de façon synchronisée, c'est-à-dire dans la même opération transactionnelle.

Il est intéressant de noter que nous n'avons pas représenté, dans ce modèle du domaine, de composant permettant de visualiser et manipuler les graphes. En effet, nous estimons que ce composant n'est qu'une aide à l'utilisation d'éléments du domaine, et non une partie intégrante de ce dernier.

Les classes sont ici volontairement simples : toutes les méthodes et tous les attributs n'y sont pas représentés. C'est grâce à l'analyse et à la conception que les classes définitives[4] seront modélisées. Ce modèle sert à donner une vision de base des éléments significatifs du domaine traité par le projet.

[4] Si tant est qu'on puisse parler de classes « définitives » dans le cadre du développement logiciel, où les applications font presque toujours l'objet de maintenances, d'améliorations et d'extensions.

5

La phase d'élaboration

La démarche méthodologique – Le modèle des cas d'utilisation – Le modèle d'analyse – L'architecture de référence

Ce chapitre traite de la phase d'élaboration. On y trouve les résultats de la capture des besoins concernant le projet traité, en utilisant la technique des cas d'utilisation, et une description de l'architecture de référence du système.

5.1 La démarche méthodologique

La phase d'élaboration pour notre projet n'a demandé qu'une seule itération. Celle-ci a comporté des activités d'expression des besoins, d'analyse et de conception.

5.1.1 Formuler et affiner la plupart des besoins

La capture des besoins est une opération extrêmement importante. En effet, ce sont les besoins qui spécifient ce que doit faire le système ; on cherche donc à s'assurer de ne capturer que les besoins réels des utilisateurs !

Les opérations de l'activité de capture des besoins suivent le schéma suivant :

- Recenser les besoins potentiels (opération déjà effectuée lors de la phase de création, voir « 4.3 : la liste des caractéristiques souhaitées ») ;
- Comprendre le contexte du système (opération déjà effectuée lors de la phase de création, voir « 4.6 : le modèle du domaine ») ;
- Appréhender les besoins fonctionnels ;
- Appréhender les besoins non fonctionnels.

Pour identifier directement les besoins du système, nous nous servons de la technique, maintenant bien connue, des cas d'utilisation.

Pour rappel, « un cas d'utilisation représente une façon d'utiliser le système par un utilisateur. Par conséquent, si on peut décrire tous les cas d'utilisation exigés par les utilisateurs, on saura exactement ce que doit faire le système. » [Jacobson I., et al, 2000]

Notons également que certaines catégories de besoins non fonctionnels ne sont pertinentes que pour certains cas d'utilisation et doivent donc leur être liées. Elles seront donc décrites dans les description des cas d'utilisation dans une section à part.

On distingue les exigences de vitesse, de disponibilité, de précision, de temps de réponse, d'usage de la mémoire, etc.

Une fois que tous les cas d'utilisation ont été identifiés, nous avons décrit complètement les divers acteurs ayant des rôles dans ces cas d'utilisation. Nous avons également tenu à jour un glossaire permettant de fixer le sens des divers mots et expressions du modèle.

Nous avons choisi de hiérarchiser le modèle des cas d'utilisation en paquetages ; ceux-ci sont décrits et justifiés dans le reste de ce chapitre.

5.1.2 Développer l'architecture de référence

On cherche à obtenir une architecture de base pour le système qui soit stable et évolutive. Les phases de construction qui suivront ne feront que « muscler » l'ossature que représente l'architecture de référence.

Pour cela, nous avons examiné les cas d'utilisation décrits. Parmi cette masse, nous avons dégagé ceux qui nous semblent importants sur le plan architectural ; ensuite, nous avons assigné un ordre de priorité à tous les cas d'utilisation. Cette notion de priorité prendra son sens plus tard, lors des diverses itérations de la phase de construction.

Les cas d'utilisation importants sur le plan architectural ont alors été analysés. On en tire un modèle d'analyse de base, et de là nous effectuons les activités de conception pour en tirer le modèle de conception de base.

5.2 Le modèle des cas d'utilisation

Le modèle complet des cas d'utilisation est constitué de plusieurs éléments : un schéma global des cas d'utilisation hiérarchisé en paquetages de cas d'utilisation, une description détaillée de chaque cas d'utilisation incluant également les exigences non fonctionnelles, une description des acteurs identifiés, un glossaire, et l'assignation de priorités aux cas d'utilisation.

5.2.1 Le modèle global

Les cas d'utilisation ont été rassemblés dans divers paquetages de cas d'utilisation, agencés de façon hiérarchique. Le modèle complet se compose de deux paquetages.

Le premier paquetage contient tous les cas d'utilisation où l'utilisation du système se fait par programmation, sans aucune utilisation interactive de la part d'un utilisateur externe. Nous l'avons encore divisé en deux paquetages : un paquetage de service contenant des cas d'utilisation utilisés à la fois par le système et par le client, et un paquetage contenant les autres opérations.

Le second paquetage général contient tous les cas d'utilisation où il y a une interactivité entre l'utilisateur et le système.

Les figures suivantes (de 5-1 à 5-8) résument notre propos :

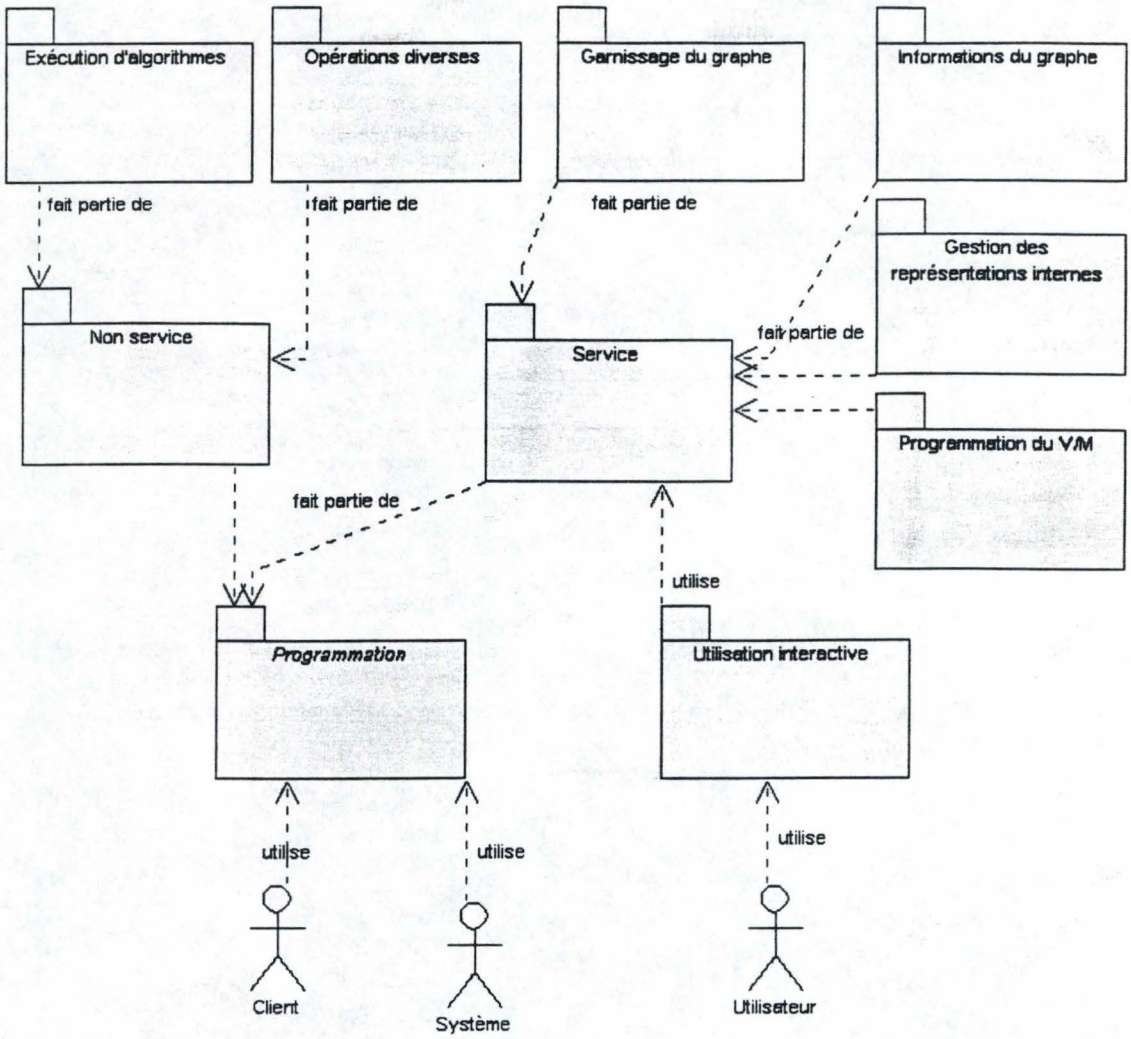


Figure 5-1 : Modèle global des cas d'utilisation, hiérarchisé en paquetages

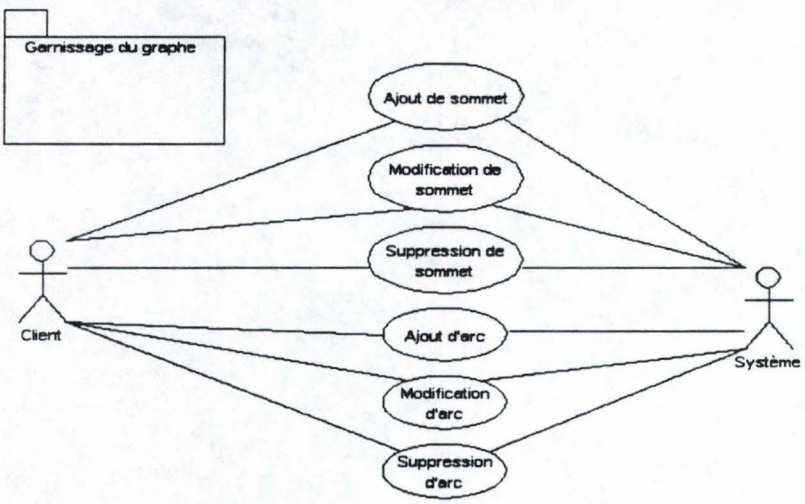


Figure 5-2 : Contenu du paquetage « garnissage du graphe »

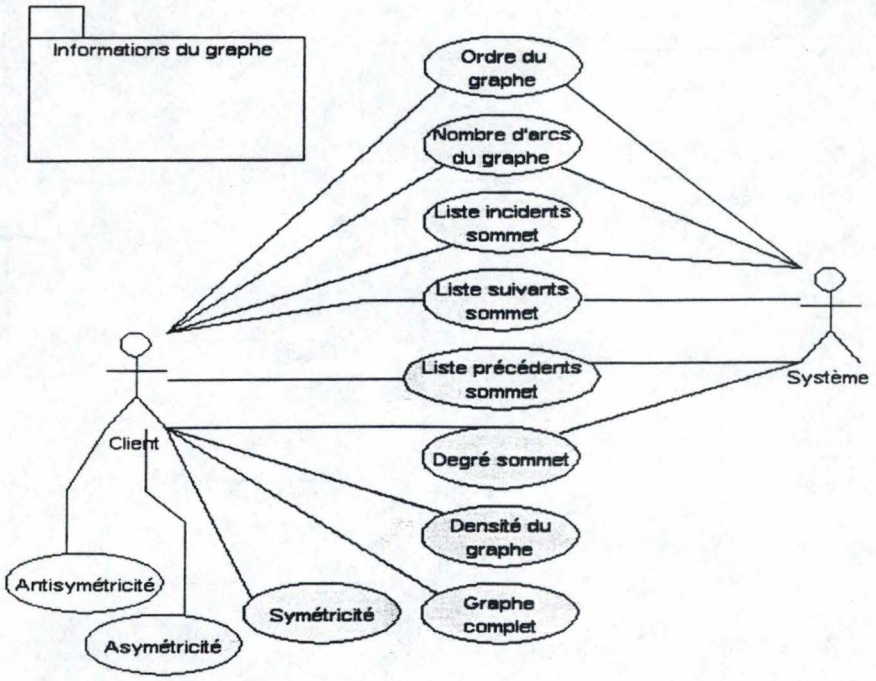


Figure 5-3 : Contenu du paquetage « Informations du graphe »

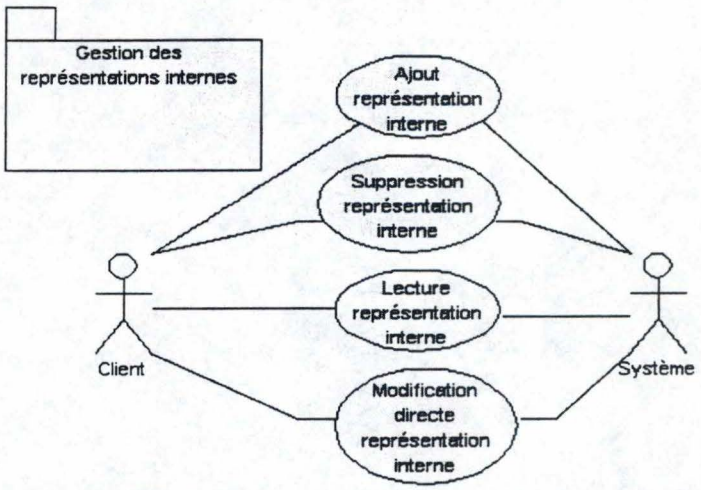


Figure 5-4 : Contenu du paquetage « Gestion des représentations internes »

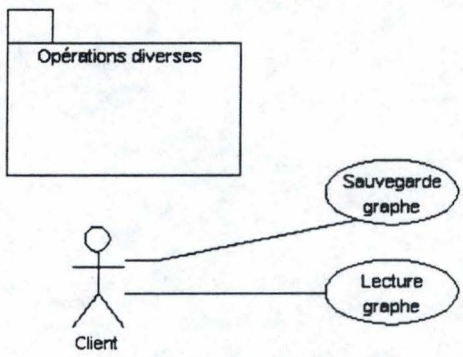


Figure 5-5 : Contenu du paquetage « Opérations diverses »

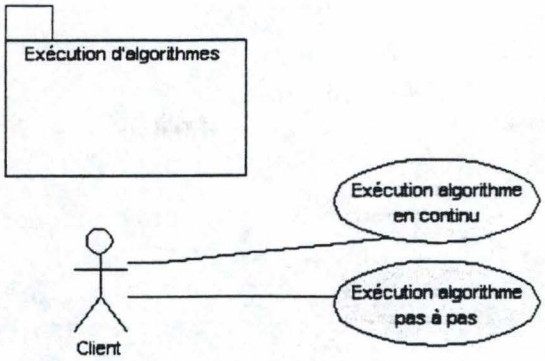


Figure 5-6 : Contenu du paquetage « Exécution d'algorithmes »

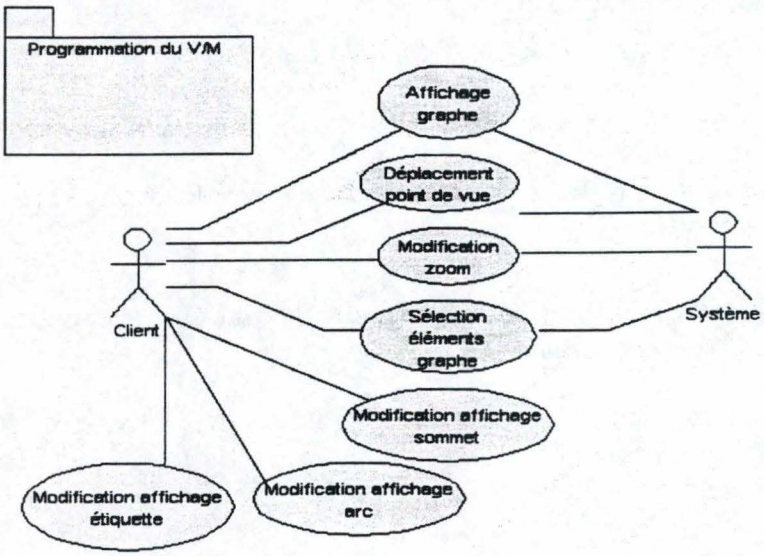


Figure 5-7 : Contenu du paquetage « Programmation du V/M »

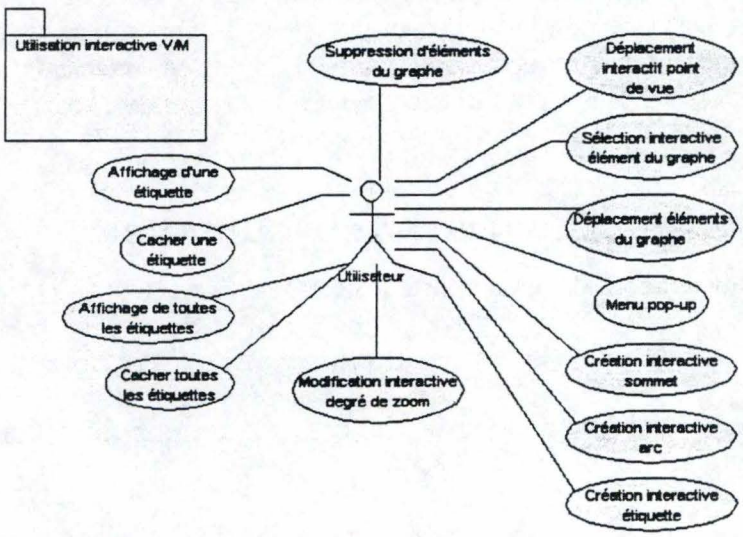


Figure 5-8 : Contenu du paquetage « Utilisation interactive du V/M »

5.2.2 Les cas d'utilisation détaillés

L'ensemble des cas d'utilisation et de leurs scénarios prenant une place beaucoup trop importante dans le cadre de ce travail, le lecteur trouvera ces cas d'utilisation (diagrammes et explications) à l'annexe 3.

5.2.3 Les acteurs

Les acteurs suivants ont été identifiés :

Le client

Le client est un développeur (analyste, programmeur, etc.) qui va utiliser, dans les applications qu'il va développer, les fonctionnalités offertes par les composants qui seront créés dans le cadre du projet. On considère qu'il initie les cas d'utilisation le concernant quand le programme qu'il développe appelle une méthode d'un composant du système.

Ses besoins se limitent à tout ce qui tourne autour de la programmation : création automatisée de graphes, garnissage de ces graphes, application d'algorithmes sur les graphes et utilisation de leurs résultats, extension des fonctionnalités du système, affichage programmé de graphes dans le visualisateur/manipulateur…

Il ne s'occupe pas d'opérations interactives avec le système.

L'utilisateur

L'utilisateur est la personne physique qui va utiliser les applications développées par le client (voir ci-dessus).

Ses besoins tournent autour des opérations interactives offertes par le composant visualisateur/manipulateur : garnissage interactif du graphe, déplacement d'éléments, déplacement du point de vue, etc.

Le système

Le système représente l'ensemble des composants créés dans le cadre du projet. Dans certains cas, un composant peut appeler les fonctionnalités d'autres composants automatiquement, sans que le client doive le programmer. Il est donc juste de considérer le système comme un acteur à part entière, car il va initier certains cas d'utilisation.

5.3 Le modèle d'analyse architecturale

Notre modèle d'analyse comporte les éléments suivants :

- La description de l'ordre de priorité assigné aux cas d'utilisation ;
- Les réalisations-analyse de cas d'utilisation importants du point de vue architectural ;
- Les analyses des classes déterminées dans les réalisations-analyse de cas d'utilisation.

5.3.1 La description de l'ordre de priorité assigné aux cas d'utilisation

L'assignation de priorités aux cas d'utilisation nous a permis de définir 5 grandes itérations de conception / développement. Nous justifions nos choix ci-dessous :

Itération 1

Cette itération va permettre de créer la base de l'architecture pour le système. A la fin de l'itération, le client pourra créer des graphes, les garnir et les lire, en utilisant plusieurs types de représentations internes. Il pourra également étendre le système en créant de nouvelles représentations internes s'il le souhaite.

De plus, on demande que le système d'événements soit mis en place directement.

Ce choix est trivial : le reste des itérations va se baser sur les résultats de cette itération. En effet, comment concevoir, réaliser et tester un composant V/M sans avoir de graphes utilisables, et dont les spécifications et interfaces sont claires ?

Le système d'événement est créé dans cette première itération afin d'avoir la base d'architecture nécessaire pour créer le reste des éléments du projet ; en effet, les algorithmes, le V/M et les autres composants à développer vont déclencher eux aussi des événements, et il est donc logique de fournir très tôt l'architecture à utiliser pour ces fonctionnalités.

Les cas d'utilisation qui devront être réalisés dans le cadre de cette première itération sont : CU001, CU002, CU003, CU004, CU005, CU006, CU014, CU015, CU016, CU018, CU019, CU020 et CU021.

Itération 2

La seconde itération a pour but d'ajouter les opérations restantes au composant Graphe et d'autres éléments divers (comme les chemins) au résultat de la première itération.

En ce qui concerne les autres éléments divers créés dans cette itération, on constate qu'ils sont assez simples et ne pénalisent pas le planning de l'itération ; de plus, certains algorithmes vont avoir besoin de tels éléments dans le cadre de leur fonctionnement, et on préfère donc les fournir dès maintenant.

Les cas d'utilisation à réaliser dans le cadre de cette seconde itération sont : CU007, CU008, CU009, CU010, CU011, CU012, CU013, CU017, CU022 et CU023.

Itération 3

La troisième itération s'occupe entièrement du système d'algorithmes. Quand elle se terminera, le client pourra appliquer des algorithmes à des graphes dans ses applications ; il pourra utiliser les résultats finaux de ces algorithmes, mais aussi employer le système d'événements pour décomposer le fonctionnement de ces algorithmes et utiliser des résultats intermédiaires[5]. Il pourra également étendre le système en créant de nouveaux algorithmes.

On peut arguer du fait d'avoir choisi de développer les algorithmes avant le composant visualisateur/manipulateur. Notre approche permet d'avoir un système complet dès la troisième itération, permettant d'offrir un traitement complet de

[5] Rappelons que l'un des buts du projet entier, dans le cadre de la proposition, est de permettre de développer des applications à caractère didactique concernant la théorie des graphes.

graphes dans le processus d'une application : création et garnissage d'un graphe, application d'algorithmes sur ce graphe, traitement des résultats.

L'autre approche, à savoir la création du composant visualisateur/manipulateur avant le système d'algorithmes, permet de créer des applications proposant une interactivité plus poussée de l'utilisateur. Cependant, nous avons estimé que le fait de ne pas pouvoir exécuter d'algorithmes avant la fin du projet est pénalisante et peu utile dans le cadre d'une utilisation à caractère didactique, d'où notre choix.

Les cas d'utilisation à réaliser dans le cadre de cette itération sont : CU024 et CU025.

Itération 4

La quatrième itération se terminera avec la création de la partie « visualisation » du composant V/M. Le client pourra alors afficher des graphes dans ses applications grâce à ce composant, modifier par programmation les types de représentation des divers éléments du graphe, et même étendre le système en ajoutant de nouveaux types de représentations d'éléments du graphe.

Le fonctionnement des algorithmes sera également complètement relié au composant, et l'utilisateur pourra voir leurs effets directement sur le graphe.

Les cas d'utilisation qui seront réalisés dans le cadre de cette itération sont : CU026, CU027, CU028, CU029, CU030, CU031 et CU032.

Itération 5

La dernière itération va fournir les fonctionnalités de manipulation du graphe du composant V/M. Quand elle sera complétée, l'utilisateur sera capable de manipuler complètement un graphe affiché par le composant V/M : déplacement du point de vue, zoom, déplacement d'éléments, garnissage interactif du graphe.

Les cas d'utilisation à réaliser dans cette dernière itération sont : CU033, CU034, CU035, CU036, CU037, CU038, CU039, CU040, CU041, CU042, CU043, CU044 et CU045.

5.3.2 Les réalisations-analyses de cas d'utilisation

Les cas d'utilisation suivants ont été déterminés comme ayant une grande importance sur le plan architectural. Ils ont donc été analysés en premier, afin d'en tirer l'architecture de référence :

- CU001 — CU006 : ils indiquent ce qu'on veut trouver dans un élément du graphe, et permettent de garnir le graphe. Ces cas d'utilisation sont donc très importants, car ils constituent la base du système.

- CU018 — CU021 : les diverses représentations internes et leur gestion sont à la base de tout le système. Il faut intégrer ce support dès le début.

De plus, rien qu'en parcourant les cas d'utilisation, on découvre en filigrane un sous-système global qu'il faut implémenter également dès le début : le système des événements. En effet, beaucoup de cas d'utilisation génèrent des événements.

Le lecteur trouvera à l'annexe 4 la réalisation-analyse de tous les cas d'utilisation du projet, dont ceux-ci.

On détermine donc 5 classes d'analyse importantes :

- Une classe représentera un graphe ;
- Une classe représentera une représentation interne. Les graphes comprennent au moins une représentation interne ;
- Une classe représentera un sommet ;
- Une classe représentera un arc ;
- Une classe représentera les informations d'habillage du graphe. Par « information d'habillage », on entend les caractéristiques permettant d'afficher les divers éléments du graphe grâce à un composant V/M ; on retrouve des informations telles que le type d'affichage d'un sommet, la couleur d'un arc, la fonte d'une étiquette, etc. On y trouvera également les emplacements des éléments dans le plan.

De plus, il y a un sous-système utilisé par toutes les classes : le sous-système des événements.

La figure 5-9 résume la vue architecturale :

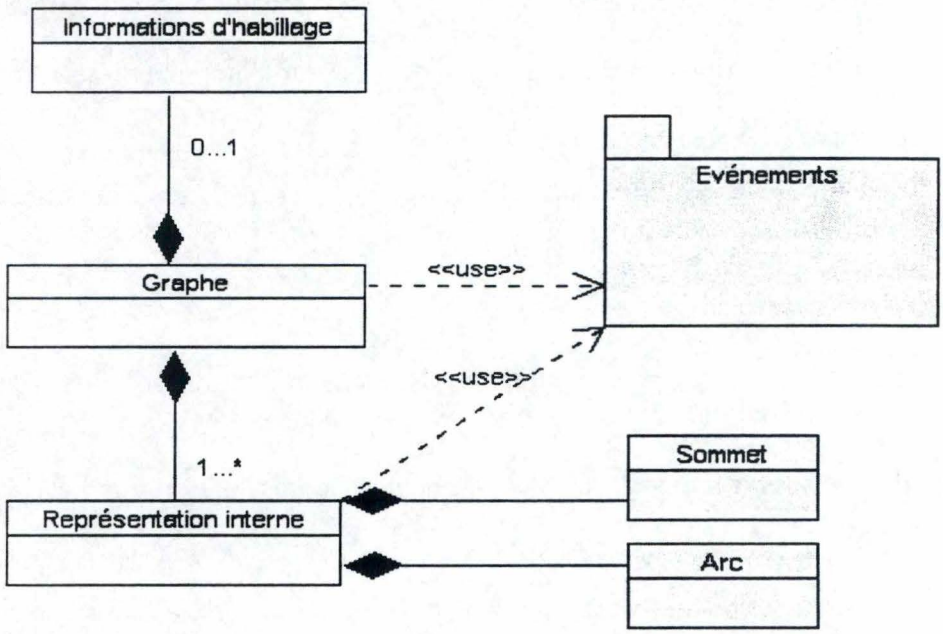


Figure 5-9 : Vue architecturale de l'analyse

5.3.3 Les analyses de classes

Suite aux réalisations-analyses de cas d'utilisation, nous avons esquissé un certain nombre de classes d'analyse. Nous allons maintenant les analyser plus en détail.

La classe « Graphe »

Elle possède les **attributs** suivants : Orientation (le graphe est-il orienté ou pas ?), Pondération des sommets (les sommets du graphe sont-ils pondérés ou pas ?), Pondération des arcs (les arcs du graphe sont-ils pondérés ou pas ?), Prochain sommet (numéro du prochain sommet qui sera créé lorsqu'on laisse le système déterminer lui-même son numéro interne).

Elle a les **responsabilités** suivantes au niveau architectural : Créer un sommet, Modifier le poids d'un sommet, Modifier l'étiquette d'un sommet, Supprimer un sommet, Créer un arc, Modifier le poids d'un arc, Modifier l'étiquette d'un arc, Supprimer un arc, Créer les informations d'habillage d'un sommet, Modifier les informations d'habillage d'un sommet, Supprimer les informations d'habillage d'un sommet, Créer les informations d'habillage d'un arc, Modifier les informations d'habillage d'un arc, Supprimer les informations d'habillage d'un arc, Créer les informations d'habillage d'une étiquette, Modifier les informations d'habillage d'une étiquette, Supprimer les informations d'habillage d'une étiquette, Renvoyer le type d'erreur qui vient de se produire.

La classe a des **relations** avec les classes d'analyse suivantes : Représentation interne, Informations d'habillage. Elle utilisera également le sous-système des Evénements.

La classe « Représentation interne »

Cette classe possède les **responsabilités** suivantes : Ajouter un sommet, Modifier le poids d'un sommet, Modifier l'étiquette d'un sommet, Supprimer un sommet, Ajouter un arc, Modifier le poids d'un arc, Modifier l'étiquette d'un arc, Supprimer un arc.

Elle a des **relations** avec les classes d'analyse suivantes : Sommet, Arc, Graphe.

La classe « Sommet »

Elle possède les **attributs** suivants : Numéro interne, Poids, Etiquette.

Elle a des **relations** avec la classe d'analyse Représentation interne.

La classe « Arc »

Elle possède les **attributs** suivants : Numéro interne du sommet d'origine, Numéro interne du sommet de destination, Poids, Etiquette.

Elle a des **relations** avec la classe d'analyse Représentation interne.

La classe « Information d'habillage »

Elle possède les **attributs** suivants : Informations d'habillage standards pour les sommets (incluant les informations d'habillage pour les sommets en surbrillance), Informations d'habillage standards pour les arcs (incluant les informations d'habillage pour les arcs en surbrillance), Informations d'habillage standards pour les étiquettes (incluant les informations d'habillage pour les étiquettes en surbrillance).

Elle a les **responsabilités** suivantes : Créer les informations d'habillage d'un sommet, Modifier les informations d'habillage d'un sommet, Supprimer les informations d'habillage d'un sommet, Créer les informations d'habillage d'un arc, Modifier les

informations d'habillage d'un arc, Supprimer les informations d'habillage d'un arc, Créer les informations d'habillage d'une étiquette, Modifier les informations d'habillage d'une étiquette, Supprimer les informations d'habillage d'une étiquette.

Elle a des **relations** avec la classe d'analyse Graphe.

5.4 Le modèle de conception architecturale

5.4.1 Diagrammes de classes

Notre modèle de conception, dans sa vue architecturale, est le suivant :

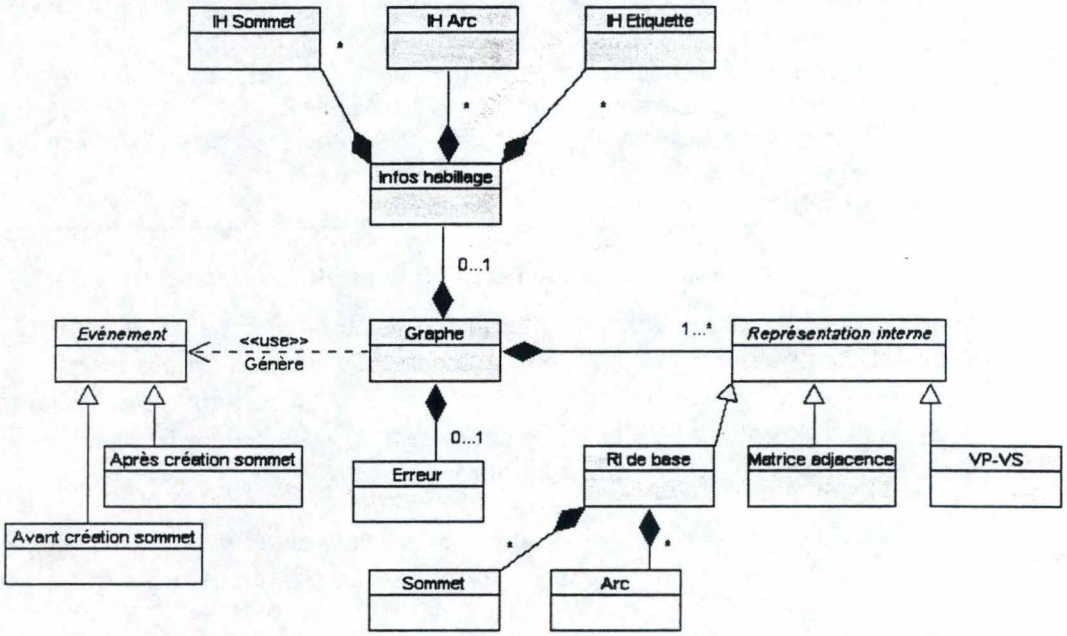


Figure 5-10 : Vue architecturale du modèle de conception.

5.4.2 Explications sur la conception

Le composant « Graphe »

Tout d'abord, on constate que l'idée de base du système est que le client utilise uniquement le composant Graphe pour effectuer ses opérations. Ainsi, les opérations de garnissage du graphe se feront normalement via l'ensemble des méthodes définies dans la classe décrite dans le diagramme 5-1 :

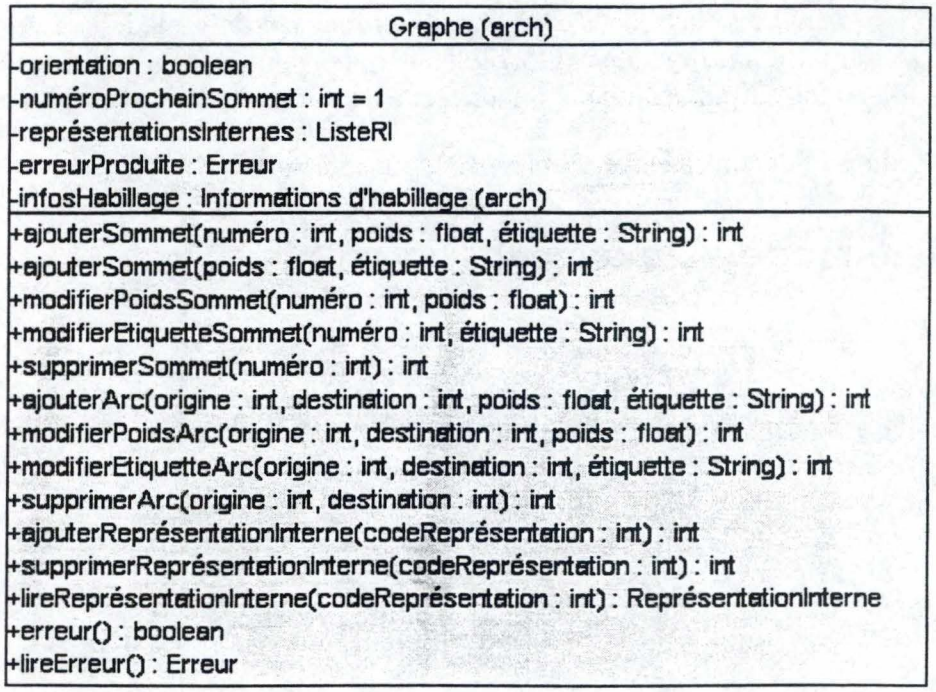


Diagramme 5-1 : Esquisse architecturale de la classe Graphe

Le composant possède également une instance de la classe *Erreur* (voir plus loin) qui indique l'éventuelle erreur qui s'est produite lors de la dernière opération effectuée. Une méthode *erreur()* indique si une erreur s'est produite ; une autre méthode *lireErreur()* renvoie la référence à cette instance d'erreur, afin que le client (ou le système) puisse utiliser ses informations.

Enfin, ses diverses représentations internes prévues dans la composition vue au point 5.4.1 sont réalisées par une liste liée ordonnée. Les point suivants explicitent cela.

Notons que cette classe n'est pas complète : elle ne représente que la vue architecturale, et sera complétée par la suite, lors de la phase de construction !

Le composant « Représentation interne »

Il y a plusieurs types de représentations internes ; d'autres représentations pourront être implémentées dans le système ultérieurement, ce qui permet d'étendre le système. La solution pour réaliser cela est d'utiliser la généralisation : on spécifie une classe abstraite *RepresentationInterne*, et on réalisera les opérations dans des classes qui hériteront de cette classe abstraite parent (par exemple, une classe « RIMatrice » pour une représentation sous forme de matrice d'adjacence).

Le diagramme 5-2 contient la description sommaire de cette classe :

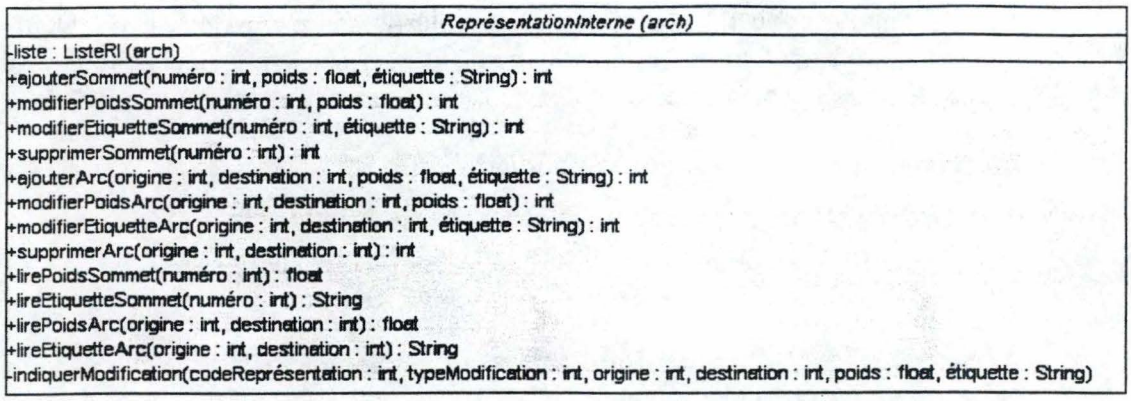


Diagramme 5-2 : Esquisse architecturale de la classe Représentation interne

Le fait d'avoir une référence « *liste* : *ListeRI* » indique qu'on désire pouvoir accéder au possesseur de la représentation interne, représenté ici par la liste des représentations internes ; cela est nécessaire pour pouvoir indiquer à cet agrégat de représentations internes une modification directe. C'est également la raison pour laquelle on a une méthode *indiquerModification()* : non abstraite, elle s'occupe de notifier la modification directe à cette liste.

Le composant « Liste des représentations internes »

Vu que le graphe pourra avoir de une à plusieurs représentations internes, mais qu'on ne peut pas dire le nombre exact à présent, on propose de concevoir la composition Graphe – Représentations internes via un composant privé que nous nommerons *ListeRI*. Ce composant comprendra une liste liée de représentations internes, identifiées uniquement par leur code. Ce code, sous forme de nombre entier, sera fourni par les représentations internes via une méthode qu'elles implémenteront.

Afin de prévoir le concept de mise à jour automatique suite à une modification directe dans une représentation interne donnée, le composant privé *ListeRI* possèdera une opération permettant d'appliquer la modification à toutes les représentations internes de sa liste ; évidemment, cette modification ne devra pas s'effectuer dans la représentation interne qui a vu la modification initiale !

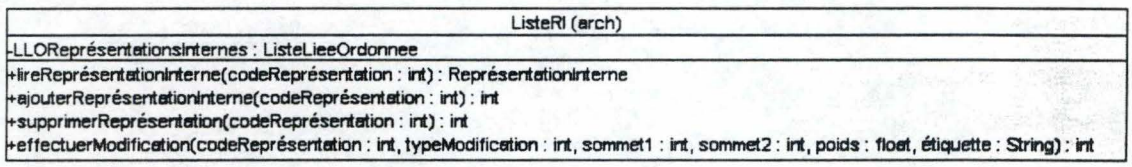


Diagramme 5-3 : Esquisse architecturale de la classe Liste des représentations internes

Les événements

Le système de gestion d'événements est maintenant assez bien connu, et implémenté directement dans beaucoup de langages de programmation « Orientés Objet ». On peut donc considérer dans notre modèle qu'il n'est pas utile de développer ce sous-système. Nous assumons également, ce travail étant normalement destiné à être compulsé par un public averti, que le lecteur connaît ce concept d'événements et comprend le principe de la programmation événementielle.

Résumons cependant très rapidement le sujet : un événement est un objet créé par le système qui permet au programmeur, s'il le désire, de réagir à cet événement en

spécifiant du code à exécuter ; s'il ne fournit pas de code, l'événement n'est tout simplement pas traité. Généralement, l'instance de l'événement fournit des informations supplémentaires utiles au traitement de l'événement.

Le composant « Informations d'habillage »

Ce composant comprend tout d'abord une description des informations d'habillage standards pour ce graphe, à la fois pour les éléments non sélectionnés et les éléments sélectionnés (qui sont affichés « en surbrillance »).

Ensuite, il comprend les informations d'habillage spécifiques à chaque sommet, chaque arc et chaque étiquette. On va représenter cela via des listes liées ordonnées, une pour chaque élément.

La classe *informations d'habillage* est conçue dans la description suivante :

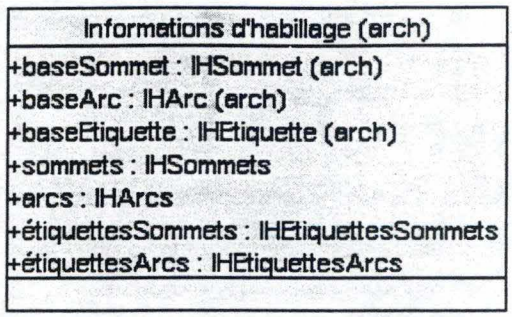


Diagramme 5-4 : Esquisse architecturale de la classe des Informations d'habillage

Les autres classes que l'on trouve dans cette description sont :

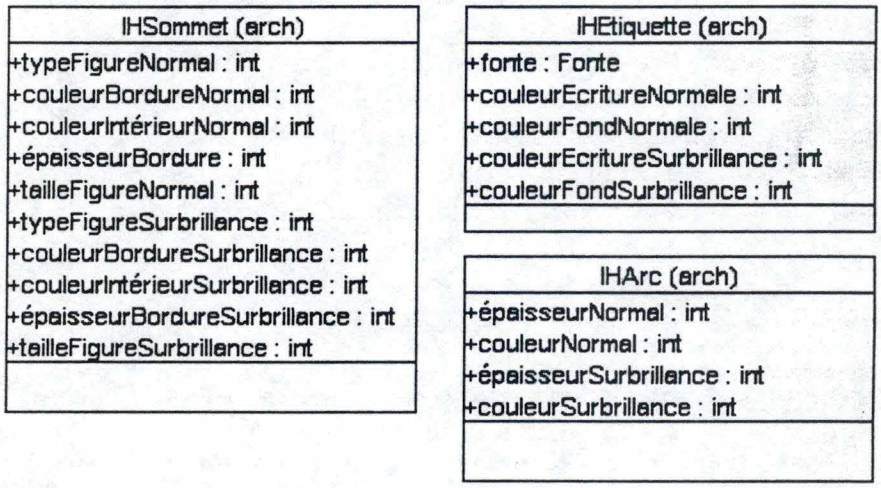


Diagramme 5-5 : Esquisse architecturale des classes Informations d'habillage des sommets, étiquettes et arcs.

Le diagramme de classes 5-6 indique les associations entre ces classes et explique notre point de vue de conception :

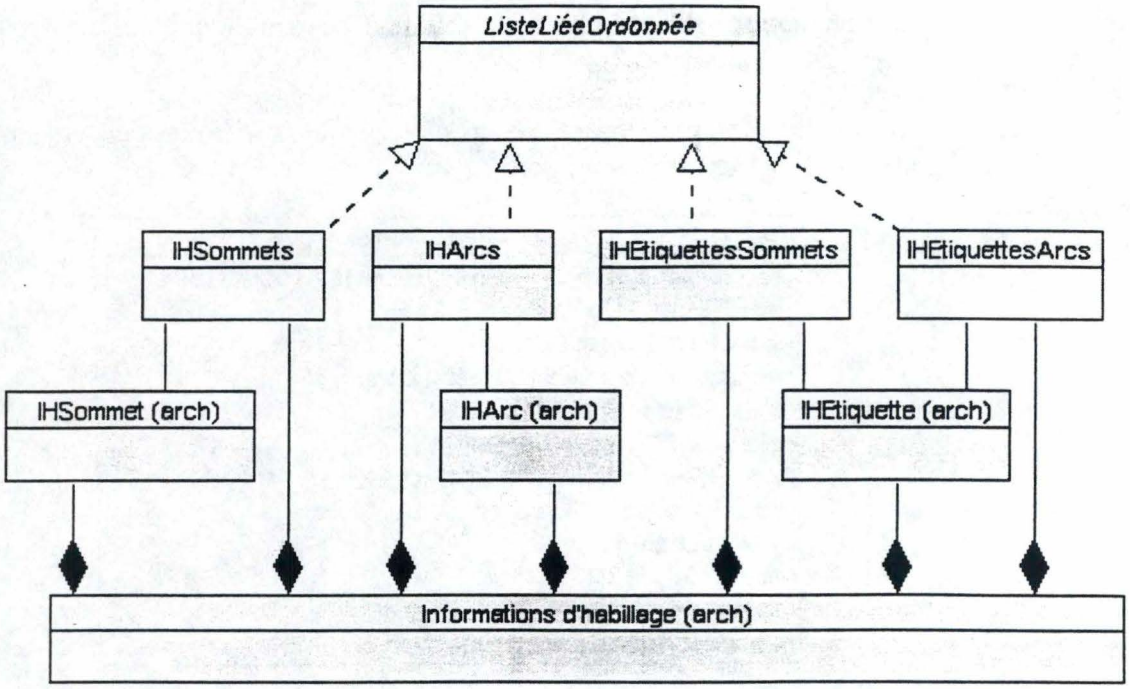


Diagramme 5-6 : Diagramme de classes indiquant la conception des informations d'habillage

En fait, la classe *Informations d'habillage* se compose à la fois des informations d'habillage de base concernant les sommets, les arcs et les étiquettes. En effet, ces informations de base représentent les informations standards qui seront appliquées à tout nouvel élément du graphe, et que le client pourra modifier à sa guise plus tard.

Cette classe comprend également une liste liée ordonnée d'informations d'habillage concernant les sommets (l'identifiant de cette liste ordonnée est le numéro du sommet), une liste liée ordonnée d'informations d'habillage concernant les arcs (l'identifiant de cette liste ordonnée est une paire de sommets), une liste liée ordonnée d'informations d'habillage concernant les étiquettes de sommets (l'identifiant de cette liste ordonnée est le numéro du sommet ayant l'étiquette) et une liste liée ordonnée d'informations d'habillage concernant les étiquettes d'arcs (l'identifiant de cette liste ordonnée est la paire de sommets de l'arc ayant l'étiquette).

Remarquons que comme les attributs de cette classe sont tous publics, on ne prévoit pas ici de méthodes permettant de garnir ces informations d'habillage : on laisse cela au composant Graphe qui possède cette classe par composition.

Le composant « Liste liée ordonnée »

Suite à ce que nous avons vu pour la classe *informations d'habillage*, on constate que le composant générique représentant une liste liée ordonnée doit :

- permettre d'identifier uniquement l'élément inséré ou cherché (car elle est ordonnée) ;
- n'accepter chaque identifiant qu'une seule fois dans la liste ;
- permettre d'utiliser un objet comme identifiant, au lieu d'une information simple (comme un nombre entier) : par exemple, si on a une liste liée

d'informations d'habillage d'arcs, il faut pouvoir identifier cet arc avec une paire de sommets.

Il s'agira donc d'un composant générique spécial à développer également ! Le diagramme 5-7 contient sa description :

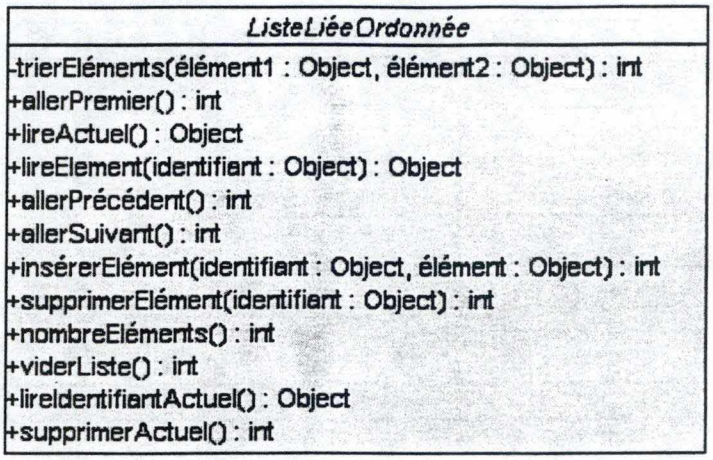


Diagramme 5-7 : Interface nécessaire pour le composant générique Liste liée ordonnée

Evidemment, si le langage de programmation offre, dans sa bibliothèque de composants de base, un composant offrant ces fonctionnalités, le développeur est fortement invité à l'utiliser.

Ce composant abstrait sera réalisé par une série de composants, lesquels devront définir le type d'éléments que la liste liée ordonnée contiendra (par exemple, des informations d'habillage de sommets), ainsi qu'une fonction indispensable pour permettre d'ordonner cette liste : *trierEléments (élément1, élément2)*. Cette fonction abstraite sera utilisée dans les opérations d'insertion, de lecture d'un élément particulier et de suppression d'un élément particulier.

Nous n'avons pas représenté ici le fonctionnement interne, cela sera fait plus tard. Remarquons juste que la liste permet un parcours dans les deux sens.

Le composant « Erreur »

Ce composant, faisant partie du composant *Graphe* général, représente une erreur qui s'est produite lors d'une opération sur le graphe. Ce système permet de fournir plus de renseignements sur l'erreur qui s'est produite ; le client n'a plus qu'à vérifier un seul code de retour pour l'opération indiquant l'erreur, puis utiliser ce composant *Erreur* s'il a besoin de plus d'informations sur ce qui s'est produit.

Le diagramme 5-8 contient la description de cette classe :

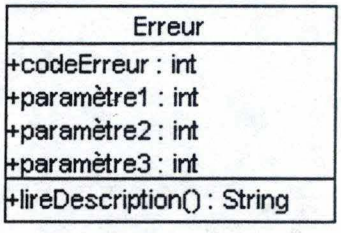


Diagramme 5-8 : Description de la classe Erreur

Lors des diverses réalisations-conception de cas d'utilisation, il faudra donc déterminer à chaque fois les erreurs qui peuvent se produire, et en dresser la liste. On pourra alors assigner un code d'erreur à chacune, et déterminer les paramètres qui l'accompagnent. Trois paramètres sont suffisants pour fournir les informations nécessaires au traitement de l'erreur ; nous proposons des nombres entiers car ces paramètres vont représenter des numéros de sommet, des codes de représentation, etc.

La méthode *lireDescription()* permettra au programmeur utilisant le système d'afficher certaines informations en cas de besoin.

Cette conception permet d'étendre le système en lui ajoutant de nouveaux types d'erreurs. Il suffit au développeur du système d'attribuer un nouveau code d'erreur, et d'ajouter la description de cette erreur dans le corps de la méthode *lireDescription()* pour pouvoir ajouter l'émission de cette erreur dans les opérations du système.

Les classes « Sommet » et « Arc »

Ces classes auxiliaires servent à représenter les informations d'un sommet ou d'un arc. Voici leurs descriptions de conception sous forme de diagrammes :

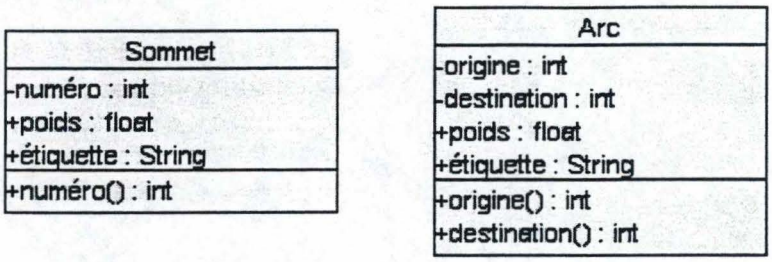


Diagramme 5-9 : Description des classes Sommet et Arc

6

La phase de construction

La démarche méthodologique – La première itération – La seconde itération – La troisième itération – La quatrième itération – La cinquième itération – Déploiement du système

Ce chapitre 6 traite de la phase de construction. C'est lors de cette phase que le produit logiciel demandé est créé. Le lecteur y trouvera donc les analyses et la conception pour les cas d'utilisation qui n'ont pas encore été abordés lors des phases précédentes, ce qui nous fournira les résultats de nos deux premiers objectifs du projet (spécification du système de modélisation des graphes, et spécification d'un composant de visualisation et manipulation).

On parlera également dans ce chapitre de l'implémentation réelle du système. On expliquera les problèmes concrets rencontrés et leurs solutions.

6.1 La démarche méthodologique

Cette phase de construction s'appuie sur l'architecture de référence qui a été définie lors de la phase d'élaboration (voir le chapitre 5).

Grâce à une série d'itérations, générant chacune un incrément, on construit peu à peu le produit logiciel demandé. La découpe en itérations a déjà été effectuée et justifiée lors de la phase d'élaboration ; nous renvoyons le lecteur à la section 5.3.1 pour plus d'informations à ce sujet.

Chaque itération va suivre la même démarche : tout d'abord, on analyse les cas d'utilisation prévus pour l'itération, puis on en tire le modèle de conception. Ensuite, on passe à l'activité d'implémentation qui va créer le code réel des composants conçus ; on effectue la planification et l'exécution des tests unitaires. Enfin, on termine l'itération par une intégration du système et la réalisation des tests d'intégration.

Il faut également remarquer que, pendant toute l'itération, il se peut que de nouveaux besoins apparaissent ; ils devront être consignés et capturés en tant que cas d'utilisation. Avant de passer à l'itération suivante, on devra planifier la construction de ces nouveaux besoins.

6.1.1 L'activité d'analyse

Il n'y a rien de neuf dans cette phase : nous allons continuer à appliquer la démarche déjà effectuée lors de la phase d'élaboration.

Chaque cas d'utilisation est analysé en détail. On en tire une « réalisation-analyse de cas d'utilisation » par scénario identifié. Les points restés en suspens lors de la capture des besoins devront être précisés ici.

Chaque réalisation-analyse de cas d'utilisation va permettre d'identifier de nouvelles classes d'analyse, ou d'étendre les responsabilités et collaborations de classes d'analyse existantes.

Remarquons qu'on ne s'occupe que de l'idée de collaboration et de responsabilité entre classes, et pas des problèmes de conception sous-jacents. Par exemple, on parle d'un ensemble d'éléments qui collaborent avec une classe, sans s'occuper du fait que cet ensemble d'éléments est représenté par une liste liée ou un tableau.

6.1.2 L'activité de conception

En nous basant sur les résultats de l'analyse, nous allons alors nous occuper de la conception de chaque cas d'utilisation.

Les classes d'analyse ont en général une traçabilité directe avec les classes de conception, mais pas toujours.

Lors de la cette activité, on va concevoir les responsabilités en classes, interfaces et méthodes, compositions, agrégats. On va également préciser les types des arguments, des valeurs de retour des méthodes ; on précise les multiplicités des éléments reliés par une association quelconque ; on précise la visibilité des divers éléments.

6.1.3 L'implémentation

Quand nous avons le modèle de conception, nous pouvons réaliser en code les besoins qui viennent d'être analysés et conçus.

Outre le code, ce qui implique la résolution d'un certain nombre de problèmes d'implémentation pure, on élabore également un modèle de déploiement : quels composants de conception font parties de quels composants d'implémentation (fichier, table, paquetage, etc.), ainsi que la distribution de ces composants sur les différents nœuds du système.

6.1.4 Les tests

On distingue les tests unitaires, et les tests d'intégration.

Les tests unitaires servent à prouver la justesse des opérations d'un composant pris séparément. Les tests d'intégration servent à démontrer que le système complet fait toujours bien ce qu'on attend de lui.

Généralement, les tests d'intégration servent de tests de non-régression lors de développements itératifs : ce qui fonctionnait à la fin de l'itération X doit toujours fonctionner de la même manière à la fin de l'itération X+1.

Cette activité commence par une planification des tests à effectuer, se poursuit par la rédaction des cas de tests et leur exécution. Enfin, il faut évaluer les résultats de ces

tests et en tirer des conclusions : l'itération est-elle acceptée et considérée comme terminée, ou faut-il revoir un certain nombre de choses ?

6.2 La première itération

Rappelons que les cas d'utilisation qui font partie de cette itération sont les suivants : CU001 (ajout de sommet), CU002 (modification de sommet), CU003 (suppression de sommet), CU004 (ajout d'arc), CU005 (modification d'arc), CU006 (suppression d'arc), CU014 (liste des suivants d'un sommet), CU015 (liste des précédents d'un sommet), CU016 (liste des sommets incidents à un sommet), CU018 (ajout d'une représentation interne), CU019 (suppression d'une représentation interne), CU020 (lecture d'une représentation interne) et CU021 (modification directe dans une représentation interne).

Cette itération représente l'implémentation effective de l'architecture de référence. Les opérations d'analyse et de conception ont déjà été effectuées en majorité lors de la phase d'élaboration (voir chapitre 5).

6.2.1 L'analyse

L'analyse a été effectuée via des réalisations-analyses des cas d'utilisation concernés. Le lecteur trouvera ceux-ci à l'annexe 4.

6.2.2 La conception

Attributs supplémentaires

Les attributs suivants de la classe *Graphe* apparaissent en filigrane des cas d'utilisation de l'itération, mais n'ont pas été cités jusqu'à présent : *sommetsPondérés* (de type boolean), *arcsPondérés* (de type boolean).

Méthodes supplémentaires

Les méthodes suivantes de la classe *Graphe* n'ont pas été citées[6] dans les cas d'utilisation, ni dans l'analyse ; elles sont cependant tellement évidentes que nous les ajoutons ici :

- *lireSommet (numéro : int) → Sommet* : renvoie un objet *Sommet* contenant une copie des données du sommet indiqué. Elle sert également à voir si un sommet existe : elle renvoie *nil* si ce n'est pas le cas, en générant une erreur.
- *lirePoidsSommet (numéro : int) → float* : renvoie le poids du sommet indiqué, ou 0 si les sommets du graphe ne sont pas pondérés.
- *lireEtiquetteSommet (numéro : int) → String* : renvoie l'étiquette du sommet indiqué.

[6] Ou alors citées comme opérations internes d'un cas d'utilisation, et pas comme méthodes à rendre disponibles.

- *lireArc (origine, destination : int) → Arc* : renvoie un objet *Arc* contenant une copie des données de l'arc indiqué. Elle sert également à voir si un arc existe : elle renvoie *nil* si ce n'est pas le cas, en générant une erreur.

- *lirePoidsArc (origine, destination : int) → float* : renvoie le poids de l'arc indiqué, ou 0 si les arcs du graphe ne sont pas pondérés.

- *lireEtiquetteArc (origine, destination : int) → String* : renvoie l'étiquette de l'arc indiqué.

Nous proposons également deux méthodes permettant de doter un graphe d'informations d'habillage, ou de l'en dépouiller :

- *créerInformationsHabillage → int* ;

- *supprimerInformationsHabillage → void*.

6.2.3 L'implémentation

Nous renvoyons le lecteur intéressé à l'annexe 5 qui spécifie complètement les classes à développer dans le système.

Création d'un graphe

Le constructeur d'un objet *Graphe* doit être soigneusement écrit. Il faut lui fournir les informations sur l'orientation du graphe, sur la pondération de ses sommets et de ses arcs. Ces informations ne seront pas modifiées par la suite.

Dans le cadre de composants réalisés pour un environnement de développement de type RAD (*Rapid Application Development*) comme Delphi, cela est problématique lorsqu'on propose un composant comme *Graphe* : il faut figer ces données non modifiables lors de la création, et donc proposer plusieurs composants différents.

Ainsi, on aura un composant « Graphe orienté, avec sommets et arcs pondérés », un composant « Graphe non orienté, avec sommets et arcs pondérés », etc. Grâce aux techniques d'héritage, on crée très vite ces composants en ne surchargeant qu'une méthode (le constructeur).

Problème : liste liée générique

Le premier petit problème qu'il reste à résoudre pour l'implémentation de cette itération concerne le composant générique *ListeLiéeOrdonnée*.

Nous avons réalisé ce composant via la technique standard : une classe représente le composant Liste voulu, et une autre classe privée contient l'identifiant de l'élément de la liste, l'élément lui-même, une référence vers l'élément précédent et une référence vers l'élément suivant. Ces références vers les éléments précédents et suivants dans la liste sont sous la forme d'un objet de même type que la classe privée.

L'élément contenu dans la liste liée sera « stocké » sous la forme d'un objet de type *Object*, à savoir l'élément de base de toute hiérarchie de composants orientés objets, et dont héritent automatiquement toutes les nouvelles classes. Dans notre cas particulier

d'utilisation du langage Delphi, cette classe *Object* existe bien, et nous n'avons donc aucun problème ; dans certains langages, comme C++ par exemple, cet objet de base n'existe pas, et il sera donc de la responsabilité du développeur du système de créer cet objet abstrait de base et d'en faire hériter tous les éléments à construire.

Cette façon de faire implique aussi que toutes les opérations de lecture des éléments devront faire l'objet d'un « *downcast* » de l'élément vers son type réel ; c'est le prix à payer pour la généricité du composant. Cependant, via la réalisation par héritage de ce composant abstrait *ListeLiéeOrdonnée*, on peut surcharger les méthodes de base nécessaires pour qu'elles renvoient l'élément sous son type réel.

Le diagramme 6-1 résume cette conception :

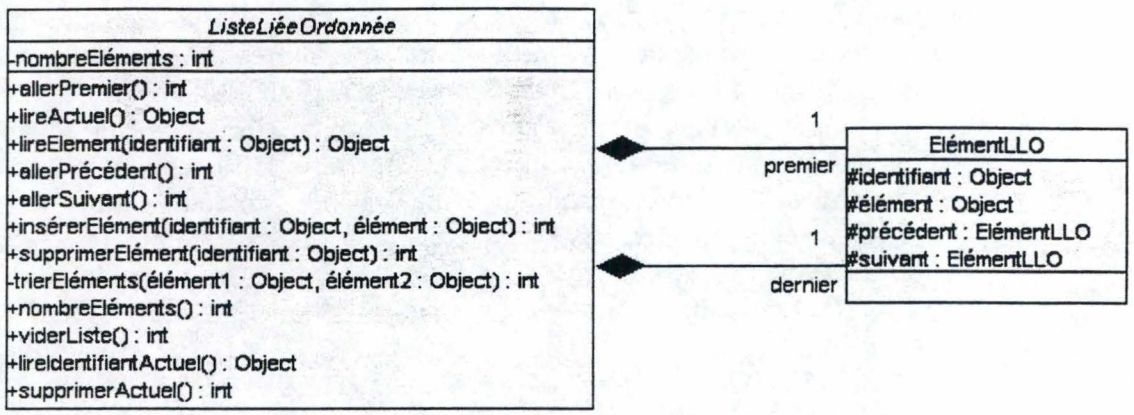


Diagramme 6-1 : Diagramme de classe décrivant la conception du composant générique Liste liée ordonnée et ses éléments

Remarquons pour terminer que la classe *ListeLiéeOrdonnée* possède un attribut appelé *nombreEléments*. Cet attribut n'est pas nécessaire du tout, puisqu'on peut facilement compter le nombre d'éléments de la liste en la parcourant dans un sens ; cependant, si la liste devient longue, il est plus rapide de mettre à jour ce compteur artificiel à chaque opération d'insertion et de suppression et de renvoyer sa valeur.

Cette liste liée ordonnée générique a été réalisée dans les composants suivants : *IHSommets*, *IHArcs*, *IHEtiquettesSommets* et *IHEtiquettesArcs*.

Chaque composant de réalisation implémente la méthode *trierElements(...)* selon le type d'identifiant.

Il surcharge également les méthodes *lireActuel*, *lireElément*, *insérerElément*, *supprimerElément* et *lireIdentifiantActuel*. Ces surcharges ne sont pas nécessaires mais sont très utiles pour une programmation plus rapide sans devoir transformer les identifiants primaires, comme un nombre entier représentant un numéro interne de sommet par exemple, en objets héritant de *Object*, et sans devoir effectuer systématiquement de *downcast* de l'élément lu vers son type réel.

Notons que ces surcharges sont permises en Delphi, notre langage de développement utilisé pour ce projet. Il se peut que certains langages n'autorisent pas de telles surcharges ; dans ce cas, il est à la charge du développeur du système d'effectuer les opérations nécessaires à l'utilisation des listes liées réalisées.

Problème : types de représentations internes à implémenter

Dans le cadre de cette implémentation, nous proposons deux types de représentations internes différentes :

- La représentation interne « maison », qui utilise une liste liée ordonnée de sommets et une liste liée d'arcs pour représenter le graphe. Elle permet d'avoir tous les renseignements voulus sur les éléments du graphe : étiquettes de chaque élément, et poids éventuel de chaque élément. Elle n'est pas efficace pour tous les usages, mais a le mérite d'utiliser le moins de place possible pour représenter le graphe, contrairement à la matrice d'adjacence.

- La représentation interne « matrice d'adjacence », qui représente un graphe sous la forme d'une matrice carrée de N*N sommets, et où chaque élément de la matrice indique la présence d'un arc (soit la présence simple, soit son poids impliquant sa présence). C'est une représentation classique, mais qui entraîne certains problèmes informatiques : beaucoup de place en mémoire si on définit préalablement la taille de la matrice, *overtime* si on permet à la matrice de grandir et diminuer dynamiquement, pas de pondération des sommets ni d'étiquettes pour eux, pas d'étiquettes pour les arcs, comment représenter un arc de poids 0 lorsque les arcs peuvent avoir tous les types de poids (positif, négatif et nul), etc.

Le but est de montrer qu'il est aisé d'ajouter des types de représentations internes au système.

Pour cela, le développeur doit :

1. Dériver une nouvelle classe à partir de la classe abstraite *RepresentationInterne* ;

2. Implémenter les fonctions de l'interface de la classe abstraite selon le type de la nouvelle représentation interne que l'on crée, en respectant les spécifications de ces méthodes ;

3. Renvoyer les erreurs et générer les événements définis dans les spécifications des méthodes de la classe abstraite ;

4. Prévoir les méthodes d'initialisation et de terminaison de la représentation interne qui seront invoquées lors de la création et de la suppression de l'instance de la classe ;

5. Assigner, dans son implémentation du système, un code à la nouvelle représentation interne ;

6. Définir et implémenter la façon dont la représentation interne « maison » peut se remplir à partir de cette nouvelle représentation interne (voir le problème suivant).

Problème : remplissage d'une nouvelle représentation interne

Pour remplir une nouvelle représentation interne, nous proposons le principe du point fixe : on passe toujours par la représentation interne « maison ». Il suffit dès lors de

prévoir la façon de remplir la nouvelle représentation interne à partir de la représentation interne « maison ».

Si cette représentation interne « maison » n'existe pas pour le graphe, on la crée temporairement ; après avoir effectué le remplissage de la nouvelle représentation interne, on efface la représentation interne « maison ».

Les inconvénients de cette implémentation sont :

1. cela peut prendre du temps lorsqu'on doit créer la représentation interne « maison » sur un gros graphe ;

2. il faut prévoir, dans la représentation interne « maison », le remplissage à partir de tous les autres types de représentations internes possibles pour ce système ;

3. il faut établir une liste de priorité entre les représentations internes à partir desquelles on crée la représentation interne « maison ». En effet, prenons l'exemple d'un graphe ayant actuellement deux représentations internes : la première ne possède pas d'informations d'étiquettes, tandis que la seconde les possède ; les étiquettes sont garnies. Si la représentation interne se base sur la première représentation interne existante, elle ne pourra pas obtenir les informations correspondant aux étiquettes, et sera donc fausse par rapport à l'existant.

L'avantage évident est de ne pas devoir créer la fonction de remplissage à partir de **tous** les types de représentation possibles, ce qui peut rapidement devenir fastidieux.

Vu que notre but est essentiellement à vocation didactique, il nous semble que l'avantage surpasse les inconvénients.

Définitions des codes d'erreur pour cette itération

Les erreurs suivantes peuvent se produire pendant les cas d'utilisation déterminés pour cette itération. A chaque erreur, on indique : son code d'erreur et le texte descriptif de l'erreur. Lorsque l'on voit un nombre entre les signes « < » et « > » dans ces textes descriptifs, il faut comprendre cela comme le numéro du paramètre de l'erreur (voir la description de la classe « Erreur » au point 5.4.2) ; la méthode *lireDescription()* de cette classe remplacera le paramètre par sa valeur réelle.

Code erreur	Texte descriptif
1	Le sommet n°<1> existe déjà.
2	Le sommet n°<1> n'existe pas.
3	Impossible de modifier le sommet n°<1>.
4	L'arc de <1> à <2> existe déjà.
5	L'arc de <1> à <2> n'existe pas.
6	Impossible de modifier l'arc de <1> à <2>.
7	La représentation interne de code <1> existe déjà.
8	La représentation interne de code <1> n'existe pas.
9	Impossible de remplir la représentation interne de code <1>.
10	La représentation interne de code <1> est la dernière représentation interne du graphe. On ne peut la supprimer !
11	Ce graphe ne possède pas d'informations d'habillage.
12	Ce graphe possède déjà des informations d'habillage.

13	Les sommets du graphe ne sont pas pondérés.
14	Les arcs du graphe ne sont pas pondérés.

Tableau 6-1 : Description des codes d'erreur générées par les opérations de la première itération

Définitions diverses pour cette itération

Enfin, on définit les codes suivants pour cette itération :

Elément	Codes	
Code de représentation interne	• grInterne = 1 (représentation « maison ») • grMatrice = 2 (représentation « matrice d'adjacence »)	
Type de modification automatique	• tmAjoutSommet = 1 • tmModifPoidsSommet = 2 • tmModifEtiquetteSommet = 3 • tmSupprSommet = 4	• tmAjoutArc = 5 • tmModifPoidsArc = 6 • tmModifEtiquetteArc = 7 • tmSupprArc = 8

Tableau 6-2 : Description des codes définis dans la première itération.

6.2.4 Les tests

Voici une description des tests réalisés sur les éléments de cette itération :

- Création de graphes selon les trois valeurs de base (orientation, pondération des sommets, pondération des arcs).
- Ajout d'une représentation interne (matrice), suivie de la suppression de la représentation interne maison, suivie de la tentative (qui doit rater) de suppression de la représentation interne sous forme de matrice d'adjacence.
- Ajout de 3 sommets (A, B et C), ajout d'arcs (A→B, B→C et tentative d'arc B→A qui doit rater si le graphe n'est pas orienté). Suppression du sommet B et vérification via les diverses méthodes (lecture directe d'un sommet ou d'un arc) qu'il reste bien 2 sommets (A et C) et aucun arc.
- Création des informations d'habillage et remplissage automatique, modification des informations d'habillage d'un sommet et d'un arc, vérification que les informations d'habillage particulières aux éléments sont bien supprimées en même temps que les éléments.
- Lecture des erreurs générées par les diverses opérations.

Pour notre facilité, nous avons développé une petite application textuelle qui effectue automatiquement ces tests en séquence, et affiche les résultats. Lorsqu'on modifie le code des objets, il suffit de la recompiler et de l'exécuter pour vérifier si tout fonctionne toujours.

6.3 La seconde itération

Rappelons que les cas d'utilisation qui sont construits lors de cette itération sont les suivants : CU007 (ordre du graphe), CU008 (nombre d'arcs du graphe), CU009 (symétricité), CU010 (asymétricité), CU011 (antisymétricité), CU012 (graphe

complet), CU013 (densité du graphe), CU017 (degré intérieur/extérieur/total d'un sommet), CU022 (sauver le graphe) et CU023 (lire le graphe).

6.3.1 L'analyse

L'analyse a été effectuée via des réalisations-analyses des cas d'utilisation concernés. Le lecteur trouvera ceux-ci à l'annexe 4.

6.3.2 La conception

L'activité de conception est assez facilement réalisée dans cette itération, car le diagramme de classes que l'on a actuellement (voir la section 5.4.1) ne change pas. Nous n'avons que de nouvelles responsabilités pour les classes existantes. Nous indiquons ici les classes qui ont été modifiées.

Modifications de la classe Graphe

On obtient les nouvelles méthodes de la classe Graphe en parcourant les réalisations-analyses de cas d'utilisation : fournir l'ordre du graphe, fournir le nombre d'arcs du graphe, indiquer si le graphe est symétrique, indiquer si le graphe est asymétrique, indiquer si le graphe est antisymétrique, indiquer si le graphe est complet, sauver le graphe, fournir la densité du graphe, fournir le degré (total, intérieur ou extérieur) d'un sommet donné.

Modifications de la classe Représentation interne

La classe Représentation interne a également été modifiée suite à l'analyse. Elle possède maintenant les méthodes supplémentaires suivantes : fournir l'ordre du graphe, fournir le nombre d'arcs du graphe, indiquer si le graphe est symétrique, indiquer si le graphe est asymétrique, indiquer si le graphe est antisymétrique, indiquer si le graphe est complet, fournir la liste des arcs impliquant un sommet donné.

Notons qu'il faudra obligatoirement créer un type particulier de Représentation interne, nommée *RIMaison*.

Enfin, le système devra proposer la méthode suivante, soit comme méthode de classe, soit comme fonction hors classe : *lire (nomFichier : String) → Graphe*.

Classe « chemin »

Nous proposons la création d'une classe *Chemin* dès maintenant, suivant le diagramme 6-2.

Elle contient la liste des arcs composant le chemin sous la forme d'une liste liée de paires de sommets.

Les opérations d'ajout et de suppression se font toujours au bout du chemin ; lors de l'ajout, on vérifie si l'origine du nouvel arc est bien la même que la destination du dernier arc du chemin actuel, et on renvoie un code d'erreur 0 si ce n'est pas le cas.

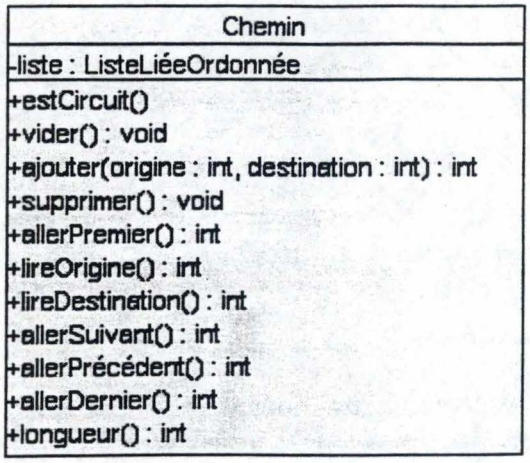


Diagramme 6-2 : conception de la classe « Chemin »

6.3.3 L'implémentation

Il n'y a aucune difficulté d'implémentation pour ces nouvelles responsabilités tirées des cas d'utilisation prévus pour cette itération.

Opérations de sauvegarde et lecture de graphes

L'opération de lecture a été réalisée grâce à une fonction incluse dans le paquetage d'implémentation « Graphe », mais ne faisant partie d'aucune classe spécifique, comme le langage de développement Delphi nous le permet. Si le projet avait été réalisé en Java, nous aurions certainement implémenté cette responsabilité sous la forme d'une fonction de classe de la classe *Graphe*, c'est-à-dire une fonction qui s'applique à la classe en elle-même et non à une instance spécifique.

Nous avons opté pour la facilité concernant la manière de décrire un graphe sous forme d'un fichier : le fichier créé est un fichier texte simple, où les éléments sont décrits séquentiellement. Chaque écriture dans ce fichier se fait dans une ligne séparée, car c'est le moyen le plus simple que Delphi propose ; on aurait pu choisir un format de type « CSV » (*Comma separated values*, ou valeurs séparées par des virgules) à la place.

Tout d'abord, on indique si le graphe est orienté (en écrivant le caractère « O » ou le caractère « N » selon que cet attribut du graphe vaut *true* ou *false*), si ses sommets sont pondérés (O ou N), si ses arcs sont pondérés (O ou N) et s'il possède des informations d'habillage ou pas (O ou N). Tout cela est nécessaire pour recréer un graphe, non encore garni, ayant les mêmes caractéristiques.

Si le graphe possède des informations d'habillage, on indique alors les valeurs standards de ces informations d'habillage, c'est-à-dire les valeurs qui seront affectées à chaque nouvel élément du graphe.

Ensuite, on écrit les informations concernant chaque sommet du graphe : chaque sommet est représenté par la lettre « S », suivie de son numéro interne, de son poids, et de son étiquette.

On fait de même pour les arcs : la lettre « A », suivie du numéro interne du sommet d'origine de l'arc, du numéro interne du sommet de destination de l'arc, de son poids, et de son étiquette.

Enfin, on traite les diverses informations d'habillage des éléments spécifiques : « IS » pour une information d'habillage de sommet, identifié par son numéro interne ; « IA » pour une information d'habillage d'arc, identifié par son sommet d'origine et son sommet de destination ; « IES » pour une information d'habillage d'étiquette de sommet, identifiée par le numéro interne du sommet auquel elle est rattachée ; « IEA » pour une information d'habillage d'étiquette d'arc, identifiée par le sommet d'origine et le sommet de destination de l'arc auquel elle est rattachée.

Il est évident que d'autres formats pourront être appliqués ; il suffit alors au développeur du système de modifier le fonctionnement de la méthode *sauver* du graphe, et celui de la fonction hors classe *lire*, afin d'implémenter un nouveau format.

Les codes d'erreurs

Les erreurs suivantes sont définies dans cette itération :

Code erreur	Texte descriptif
15	Le graphe n'est pas orienté, l'opération n'a donc aucun sens.
16	Impossible de calculer la densité du graphe, car il ne possède aucun sommet.
17	Répertoire de sauvegarde du fichier non trouvé.
18	Le fichier indiqué existe déjà.
19	Le fichier indiqué n'existe pas.

Tableau 6-3 : Description des erreurs générées par les opérations de cette itération n°2

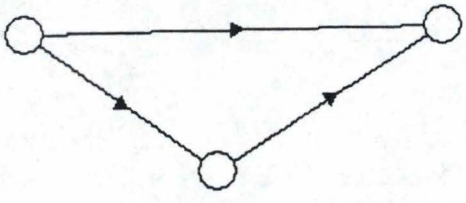
6.3.4 Les tests

Nous rajoutons les tests suivants à notre application de test :

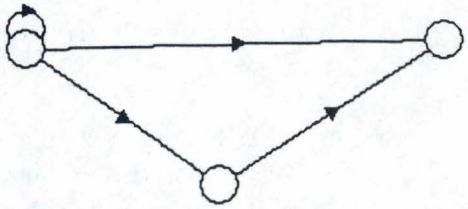
- Créations successives des graphes de la figure 6-1 et test des résultats des méthodes indiquant si ces graphes sont symétriques, asymétriques, antisymétriques et complets.

- Création du graphe de la figure 6-2 et vérification des résultats des méthodes fournissant l'ordre du graphe, le nombre d'arcs, la densité et les divers degrés (total, intérieur et extérieur) du sommet 1.

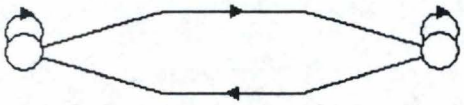
- Sauvegarde du graphe de la figure 6-2 et chargement de ce graphe, et vérification de la bonne présence de tous les éléments.



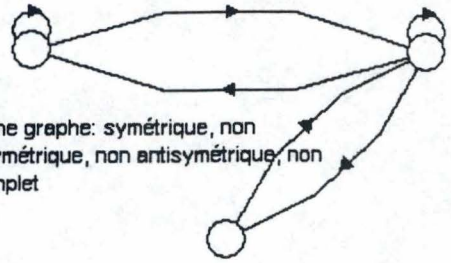
1er graphe: non symétrique, asymétrique, antisymétrique, non complet



2ème graphe: non symétrique, non asymétrique, antisymétrique, non complet



3ème graphe: symétrique, non asymétrique, non antisymétrique, complet



4ème graphe: symétrique, non asymétrique, non antisymétrique, non complet

Figure 6-1 : Graphes de test pour diverses fonctions de la seconde itération

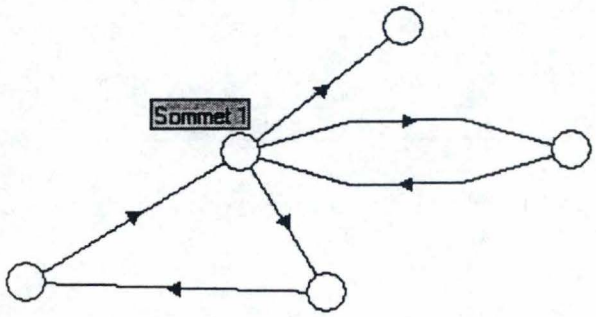


Figure 6-2 : Graphe de test pour certaines fonctions de la seconde itération

6.4 La troisième itération

Rappelons que les cas d'utilisation qui sont construits lors de cette itération sont les suivants : .CU024 (exécution d'un algorithme en continu) et CU025 (exécution d'un algorithme pas à pas).

6.4.1 L'analyse

L'analyse a été effectuée via des réalisations-analyses des cas d'utilisation concernés. Le lecteur trouvera ceux-ci à l'annexe 4.

6.4.2 La conception

Nous avons ici un gros morceau de conception : comment créer un système générique permettant de créer les algorithmes ? Nous proposons la solution suivante :

Tout d'abord, les algorithmes applicables aux graphes seront tous dérivés par héritage d'une classe abstraite appelée *algorithme*. Cette classe abstraite fournira l'interface de

base pour l'utilisation des algorithmes, tandis que les classes enfants fourniront l'implémentation des méthodes abstraites de cette interface ainsi que le fonctionnement interne de l'algorithme.

Ainsi, les événements et le fonctionnement de base seront « câblés » dans la classe abstraite, et le développeur d'un nouvel algorithme n'aura plus qu'à écrire le code correspondant à l'algorithme lui-même, sans devoir se soucier de générer les événements de base ni de comment répondre à une demande d'exécution.

L'exécution d'un algorithme s'effectuera selon le scénario commun suivant :

1. On effectue **l'initialisation** de l'algorithme : cette étape permet d'initialiser des variables, des tableaux, etc. avant le début des itérations de l'algorithme.

2. On **teste** si l'algorithme est terminé ou pas : si l'algorithme n'est pas fini, on effectue une **itération** de l'algorithme et on recommence cette étape ; si l'algorithme est terminé, on passe à l'étape suivante.

3. On effectue la **terminaison** de l'algorithme : cette étape permet de mettre à jour les derniers éléments nécessaires avant la fin de l'exécution.

Chacune de ces étapes (initialisation, test de fin, itération, terminaison) peut être vide ! Ainsi, certains algorithmes n'auront pas besoin d'étape de terminaison, par exemple, tandis que d'autres n'auront pas d'étape d'initialisation. Tout dépend de l'algorithme.

Cette idée de base permet de représenter tous les types d'algorithmes. Tout d'abord, beaucoup d'algorithmes rentrent directement dans ce schéma, étant donné qu'on peut avoir l'une ou l'autre étape vide

Ensuite, s'il arrivait qu'on doive implémenter un algorithme ne rentrant pas du tout dans ce schéma, on peut toujours y arriver artificiellement : il suffit d'écrire tout le fonctionnement de l'algorithme dans l'opération d'initialisation, d'écrire le test de fin de façon qu'il indique que l'algorithme est fini à tout moment, et d'avoir une opération de boucle et une opération de terminaison vides. Evidemment, cette solution est conceptuellement peu jolie, et implique plus de travail de la part du développeur (qui devra implémenter lui-même l'événement « boucle de l'exécution »), mais elle a le mérite d'être générique.

Le diagramme 6-2 indique la conception de la classe abstraite de base *algorithme* :

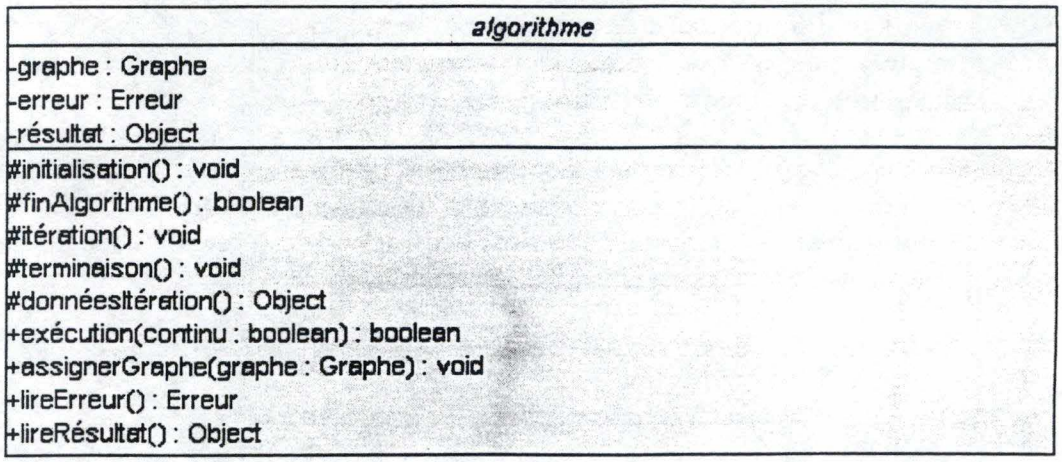


Diagramme 6-2 : Conception de la classe abstraite Algorithme

On a ici les opérations importantes de la classe : assigner un graphe à l'algorithme, l'exécuter (pas à pas ou en continu), lire l'erreur éventuelle, et lire le résultat de l'algorithme.

Les méthodes de type « protégé », comme *initialisation()*, sont abstraites : elles seront implémentées dans les classes dérivées.

Le résultat fourni par l'algorithme sera un type de données créé par le développeur, dérivé de la classe de base *Object* ; la lecture du résultat par le client devra donc se faire accompagnée d'un *downcast* depuis *Object* vers le type de données réel. Si on le désire (et si le langage de programmation le permet), on peut surcharger cette opération dans la classe enfant pour qu'elle renvoie le type réel du résultat.

Les données du fonctionnement de l'algorithme seront aussi représentées par un type de données créé par le développeur, dérivé de la classe de base *Object*. Elles seront fournies au client lors de chaque événement « boucle » de l'algorithme.

La description de la création d'un nouvel algorithme est décrite en détail dans la partie implémentation de cette itération (voir 6.4.3).

6.4.3 L'implémentation

Nous avons réalisé la classe de base abstraite *algorithme*, ainsi que deux algorithmes à partir de cette classe : Warshall (accessibilité totale d'un graphe) et Moore-Dijkstra (recherche de chemins minimaux dans un arbre).

Implémentation de la classe de base « algorithme »

La classe abstraite de base *Algorithme* suit le diagramme de spécification trouvé au point 6.4.2. Nous avons ajouté quelques méthodes générales :

- *indiquerErreur (c, p1, p2, p3 : int) → void* : cette méthode permet d'écrire un objet de type *Erreur* dans l'attribut *erreur* ; sert surtout à un accès depuis les classes héritées (car un accès privé n'est pas visible depuis une classe héritée) et à la rapidité de création d'une erreur dans l'implémentation d'un algorithme (sans devoir créer l'instance, etc.)

- *viderErreur → void* : efface l'attribut *erreur* et le met à *nil*. Même chose que pour la méthode *indiquerErreur* en ce qui concerne le pourquoi.

- *écrireRésultat (r : Object) → void* : cette méthode garnit l'attribut *résultat* avec l'objet qu'on lui passe. Sert à un accès depuis les classes héritées.

- *créationAlgorithme → void* : cette méthode abstraite sert à initialiser certaines variables lors de la création du composant algorithme. Comme les méthodes *initialisation*, *finAlgorithme*, etc., elle peut être vide s'il n'y a rien à initialiser à la création du composant.

- *construireCorrespondanceSommets → void* : cette méthode va servir à créer une table de correspondance entre les sommets réels du graphe, dont la numérotation peut ne pas être continue suite à des suppressions de sommets, et les sommets qui seront peut-être utilisés par l'algorithme, et dont la numérotation sera ininterrompue (par exemple, la liste de sommets réels peut être [1, 2, 4, 5, 6, 8, 10, 15, 16, 23] ; la fonction renverra la table de correspondance suivante : [[1, 1], [2, 2], [4, 3], [5, 4], [6, 5], [8, 6], [10, 7], [15, 8], [16, 9], [23, 10]]). C'est utile pour certains algorithmes, donc on le propose au niveau de la classe de base. Par exemple, l'algorithme de Warshall gagne en optimisation grâce à cela (voir plus bas).

- *correspondance(numéro : int) → int* : cette fonction sert lorsqu'on a une table de correspondance entre sommets, créée grâce à la méthode *construireCorrespondanceSommets*. Elle renvoie le numéro interne réel du sommet dans le graphe lorsqu'on lui fournit le numéro de ce sommet dans la numérotation (continue) de l'algorithme.

Réalisation de l'algorithme de Warshall

On commence par définir les types de données représentant les données actuelles de l'exécution du graphe :

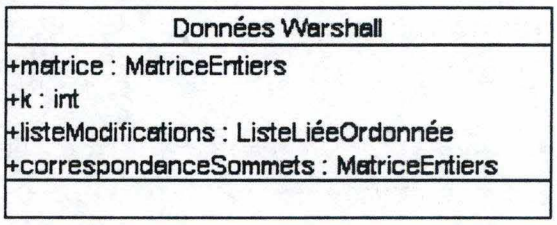


Diagramme 6-3 : Conception de la classe contenant les données de fonctionnement de l'algorithme de Warshall

Nous avons une matrice d'entiers, représentant la valeur de la matrice M (fermeture transitive du graphe). Ensuite, on a la valeur de la variable K de l'algorithme de Warshall :c'est le numéro de l'itération en cours.

Nous avons également une liste liée ordonnée contenant la liste des arcs ajoutés à chaque itération (les éléments de la liste sont donc des paires de sommets, identifiés par une simple numérotation séquentielle).

Enfin, nous avons une matrice de correspondance de sommets : cette subtilité permet d'avoir la matrice M la moins grande possible. En effet, dans le cadre d'un graphe

garni ayant subi de nombreux ajouts et suppressions de sommets, il va y avoir de nombreuses colonnes et lignes vides dans cette matrice. Nous proposons donc un simple système où la matrice d'adjacence de départ est recopiée dans la matrice M en ne tenant pas compte des sommets supprimés, et sans tenir compte des pondérations d'arcs ; dans ce cas, il faut que nous ayons un tableau permettant de donner le numéro interne réel d'un sommet par rapport au numéro du ce sommet dans cette matrice M simplifiée.

Tous ces éléments sont publics, de façon que le client qui obtient ces données dans l'événement « Boucle dans l'exécution » puisse y accéder sans problème et sans devoir utiliser de méthodes « lire...() » inutiles.

On définit ensuite le type de données représentant le résultat de l'algorithme :

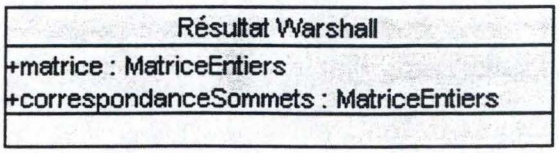


Diagramme 6-4 : Conception de la classe contenant le résultat de l'algorithme de Warshall

Cette structure est très simple et correspond aux données de fonctionnement de l'algorithme (voir la classe *Données Warshall* ci-dessus), sans les données inutiles (la variable K et les changements apportés par l'itération).

Ensuite, nous définissons la classe (contenant l'algorithme) « Warshall » :

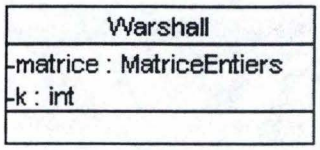


Diagramme 6-5 : Conception de la classe réalisant l'algorithme de Warshall par héritage de la classe abstraite Algorithme

Très simple ! Il nous suffit maintenant d'implémenter les méthodes abstraites de la classe *algorithme* dans cette classe *Warshall*. Nous avons fait comme ceci :

- Initialisation : on crée une matrice vide carrée de la taille de l'ordre du graphe, et on initialise la variable K de l'algorithme à 0.

- Test de fin : si K > ordre du graphe, alors l'algorithme est terminé.

- Itération : la première itération consiste en la construction de la matrice (matrice simplifiée avec table de correspondance avec les sommets réels du graphe), les autres itérations calculent les modifications de la matrice selon les spécifications de l'algorithme. Nous ajoutons la génération d'un événement lorsqu'on examine une case donnée de la matrice (aux fins d'informations pour un traitement didactique de cet algorithme), la mise à jour des modifications éventuelles de cette itération (pour que le client puisse y accéder dans le type de données *Données Warshall* lors de l'événement « Boucle dans l'algorithme »), et la génération d'un événement lorsqu'on ajoute un arc à la matrice.

- Terminaison : on profite de cette méthode pour garnir le résultat de l'algorithme avec la matrice M calculée lors des itérations de l'algorithme, et la table de correspondance des sommets.

En ce qui concerne la construction de la matrice M de l'algorithme, on propose d'ajouter au graphe la représentation interne « maison » s'il ne la possède pas actuellement, de construire la matrice M à partir de cette représentation interne maison (réalisé assez facilement), et d'effacer la représentation interne « maison » du graphe s'il ne la possédait pas auparavant.

Réalisation de l'algorithme de Moore-Dijkstra

Nous appliquons le même principe de développement que pour l'algorithme précédent.

Tout d'abord, on définit la structure de données du fonctionnement de l'algorithme :

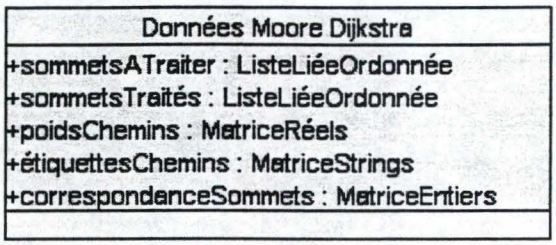


Diagramme 6-6 : Conception de la classe contenant les données de fonctionnement de l'algorithme de Moore-Dijkstra

Elle contient : la liste des sommets restant à traiter, la liste des sommets déjà traités, le vecteur des poids actuels des sommets, et le vecteur des étiquettes actuelles des sommets (contenant le numéro du sommet précédent dans le chemin de poids inférieur actuel).

Ensuite, on définit la structure de données correspondant au résultat final de l'algorithme :

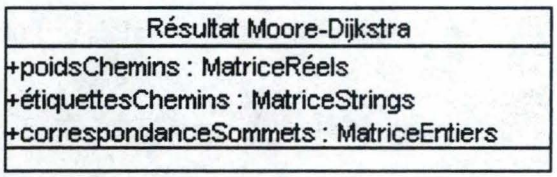


Diagramme 6-7 : Conception de la classe contenant les résultats de l'algorithme de Moore-Dijkstra

Elle correspond à la structure de données de fonctionnement, de laquelle on a enlevé les informations inutiles (liste des sommets traités et liste des sommets à traiter).

Enfin, on définit la classe « Moore-Dijkstra » héritant de la classe abstraite *algorithme* :

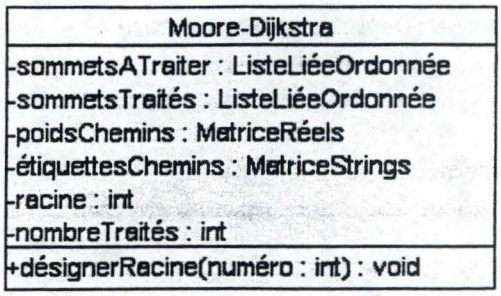


Diagramme 6-8 : Conception de la classe réalisant l'algorithme de Moore-Dijkstra à partir de la classe abstraite Algorithme

Les attributs de cette classe sont explicites : les quatre premiers correspondent aux données de fonctionnement de l'algorithme, la racine est le numéro interne du sommet que le client devra désigner comme racine de départ pour l'algorithme, et *nombreTraités* compte le nombre de sommets traités.

Remarquons que nous ajoutons une méthode à cette classe : en utilisant *désignerRacine()*, le client va remplir ses obligations de fourniture des informations initiales du graphe. L'algorithme teste si le sommet indiqué est bien une racine, indique une erreur si ce n'est pas le cas et met à jour l'attribut *racine* dans le cas contraire.

Remarquons également que nous utilisons la même technique que pour l'algorithme Warshall d'utilisation d'une table de correspondance entre les sommets réels et les sommets qui seront utilisés dans l'algorithme (renumérotés de façon continue).

Nous implémentons ensuite les méthodes abstraites de la classe *algorithme* :

- Initialisation : si on n'a pas de racine indiquée, on annule l'exécution en indiquant une erreur ; sinon on vide la liste des sommets traités, on initialise la liste des sommets à traiter, le vecteur des étiquettes des sommets avec des chaînes vides et le vecteur des poids des chemins à une constante entière appelée « INFINI » (valant 999999999 dans notre implémentation).

- Test de fin : si *nombreTraités* est égal à l'ordre du graphe, l'algorithme est terminé.

- Itération : on choisit le sommet Y non traité de poids minimum parmi les non traités (ou la racine pour la première itération), nous générons un événement supplémentaire indiquant qu'on a choisi le sommet Y (et son poids), on ajoute ce sommet à la liste des sommets traités et on le supprime de la liste des sommets à traiter, et on passe en revue les sommets non encore traités, afin de mettre éventuellement les poids à jour pour ces sommets si Y est un précédent du sommet sur un chemin plus court que le meilleur actuel (dans ce cas, on génère un événement supplémentaire indiquant qu'on a modifié le chemin pour ce sommet).

- Terminaison : On garnit le résultat de l'algorithme.

Pour le garnissage des sommets à traiter, on va également passer par la représentation interne « maison », qui sera ajoutée au graphe au besoin (et supprimée à la fin de l'exécution).

Les codes d'erreurs

Les erreurs suivantes sont définies dans cette itération :

Code erreur	Texte descriptif
20	Exécution impossible : les prérequis de l'algorithme non remplis.
21	Exécution impossible : les informations initiales sont incorrectes.
22	Problème lors de l'exécution de l'algorithme.
23	Exécution de l'algorithme interrompue par l'utilisateur.

Tableau 6-4 : Description des erreurs générées par les opérations de cette itération n°3

6.4.4 Les tests

Les tests sont réalisés dans les applications de démonstration que nous verrons dans le chapitre suivant (voir les points 7.3.1 et 7.3.2) ; ces applications refont les exemples du cours de M. Leclercq, et le test consiste à vérifier si les résultats correspondent avec ceux du cours.

6.5 La quatrième itération

Rappelons que les cas d'utilisation qui sont construits lors de cette itération sont les suivants : .CU026 (affichage du graphe), CU027 (modification de l'affichage d'un sommet), CU028 (modification de l'affichage d'un arc), CU029 (modification de l'affichage d'une étiquette), CU030 (déplacement du point de vue), CU031 (modification du paramètre de zoom) et CU032 (sélection d'éléments du graphe).

6.5.1 L'analyse

L'analyse a été effectuée via des réalisations-analyses des cas d'utilisation concernés. Le lecteur trouvera ceux-ci à l'annexe 4.

6.5.2 La conception

L'étude des réalisations-analyses des cas d'utilisation de cette itération nous amène à créer une nouvelle classe : le visualisateur/manipulateur. Il y aura également quelques adaptations à faire dans les classes existantes pour tenir compte des nouvelles responsabilités, des nouveaux attributs repérés par ces nouveaux besoins.

L'affichage

Tout d'abord, il faut comprendre que l'affichage se fait selon les axes X (axe horizontal allant de gauche à droite), et Y (axe vertical allant du haut vers le bas). Cet axe Y est dans le sens contraire au sens habituel.

La classe Visualisateur/manipulateur

Le V/M est un composant visuel, comportant une zone d'affichage couvrant toute sa surface. Sa taille peut être redimensionnée.

Le diagramme 6-9 décrit la conception de la classe Visualisateur / manipulateur :

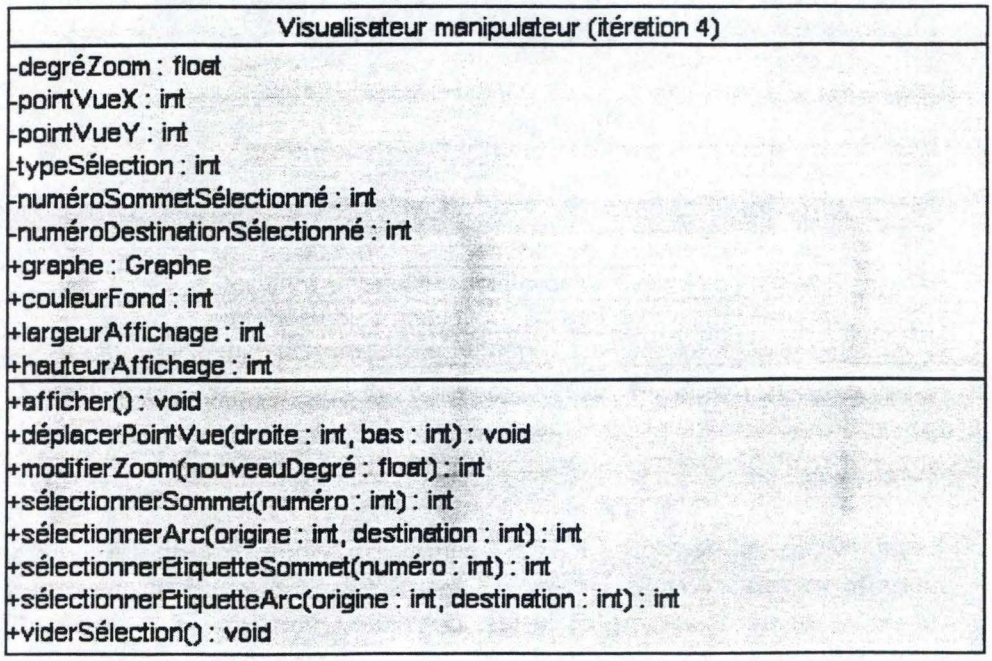


Diagramme 6-9 : Conception de la classe Visualisateur / manipulateur

Dans notre idée de la conception de cette classe, le « point de vue » du composant correspond à l'emplacement dans le plan du point central de la zone d'affichage. Une autre possibilité aurait pu être de prendre le point supérieur gauche de la zone d'affichage comme point de vue. Notre choix est motivé par le seul souci d'esthétique des opérations de zoom.

L'attribut correspondant à la couleur de fond est représenté par un entier ; selon l'environnement de développement, on pourra utiliser un code ou une énumération. Le type de sélection est également représenté par un entier ; des constantes entières permettent une meilleure lisibilité.

Les classes d'informations d'habillage

Maintenant que l'on va pouvoir afficher les éléments du graphe, il faut savoir où les placer dans la plan ! On va donc modifier les classes *IHSommet* et *IHEtiquette*. Le diagramme 6-10 (voir page suivante) indique les conceptions actuelles de ces classes.

Un sommet a maintenant un emplacement fixé dans le plan. Une étiquette (de sommet ou d'arc) a maintenant des attributs de « déplacement » : pour une étiquette de sommet, c'est le déplacement, par rapport à l'emplacement de son sommet, du coin inférieur gauche de l'étiquette ; pour une étiquette d'arc, c'est le déplacement, par rapport au milieu de l'arc, du coin inférieur gauche de l'étiquette.

Affichage des arcs

Remarquons qu'un arc n'a pas de données d'emplacement : un arc est représenté par une ligne droite entre les emplacements des deux sommets qu'il relie. Si le graphe est orienté, on trouve également un petit triangle au milieu de l'arc indiquant son sens.

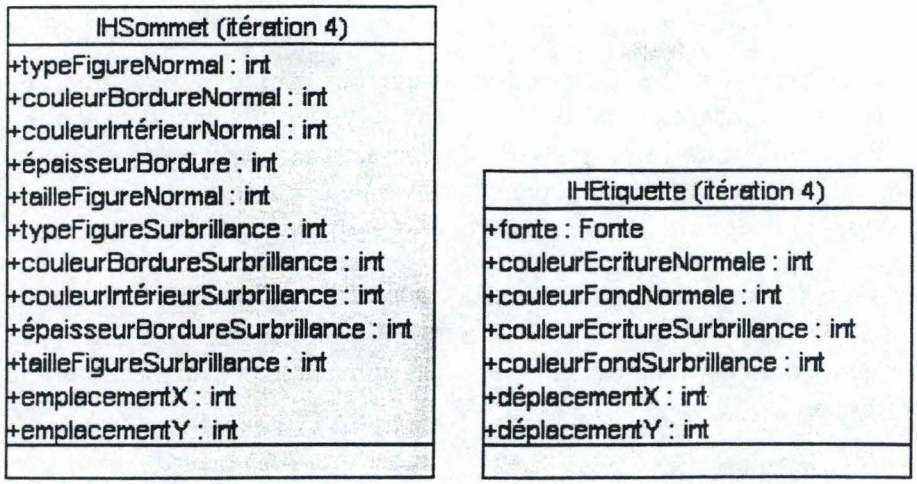


Diagramme 6-10 : Conception des classes IHSommet et IHEtiquette suite à l'itération 4

Cependant, si le graphe possède un arc dans le sens contraire, on applique une autre méthode pour plus de lisibilité : l'arc sera constitué de 3 segments de droite. Le premier segment part du sommet d'origine ; sa destination correspond au point calculé au premier tiers de la droite reliant le sommet d'origine au sommet de destination, auquel on applique un déplacement de 90 degrés (à angle droit, donc) égal à la distance totale de l'arc « standard » auquel on applique un ratio de 10%. Le segment part de ce point vers un autre point calculé au second tiers de la droite reliant le sommet d'origine au sommet de destination, auquel on applique le même déplacement ; ce segment est donc parallèle à la droite d'origine. Enfin, le dernier segment part de ce second point vers l'emplacement du sommet de destination.

On est ainsi certain, grâce au déplacement de 90 degrés, que les deux arcs en sens contraires seront bien séparés et visibles.

La figure 6-3 explique visuellement notre propos :

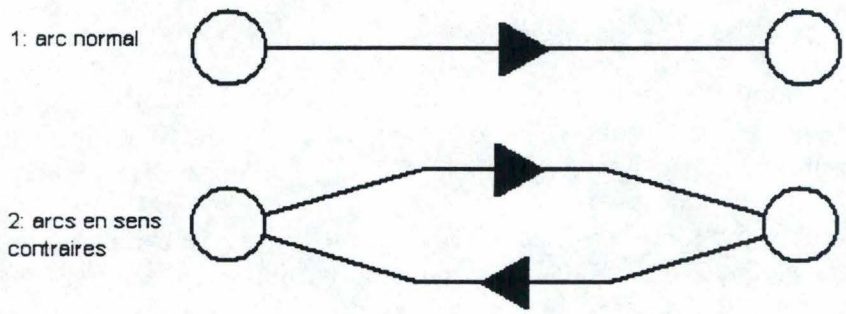


Figure 6-3 : Description visuelle de l'apparence des arcs dans notre projet

Cette façon de représenter les arcs est la plus naturelle à notre sens et la plus lisible. Cependant, il n'y a rien de prévu pour les éventuels croisements d'arcs ; il est, dans l'état actuel du projet, de la responsabilité de l'utilisateur d'arranger visuellement son graphe de façon à le rendre le plus lisible possible. Le système peut néanmoins être étendu avec des algorithmes ou des heuristiques spécifiques automatisant cette tâche, ou en insérant des « ponts » à un arc lors de croisements. Comme tout ceci est hors du *scope* initial du projet, nous laissons ces possibilités au bon vouloir de futurs développeurs.

Affichage des sommets

Nous proposons deux types d'affichage des sommets : un cercle, et un carré. L'intérêt de notre conception est de permettre au développeur qui souhaite étendre le système de créer de nouveaux types de représentations : il lui suffit de coder la nouvelle façon de représenter un sommet dans la méthode d'affichage du graphe, et de lui assigner un code (qui sera utilisé dans l'attribut « *typeFigure*... » de ce sommet).

Selon le type de figure utilisé, l'attribut « *tailleFigure*... » aura une signification différente. Pour un cercle, il correspond au diamètre en pixels standards (c'est-à-dire avant application du degré de zoom) ; pour un carré, c'est la taille d'un côté en pixels standards.

L'emplacement du sommet correspond dans les deux cas au milieu de la figure.

Affichage des étiquettes

Une étiquette sera toujours représentée sous la forme d'un rectangle horizontal (ce qui est le plus lisible et le plus facile à implémenter). L'emplacement de cette étiquette est indiqué par son **déplacement** par rapport à l'emplacement du sommet correspondant ou par rapport au milieu de l'arc correspondant. Ainsi, si on déplace un sommet, l'étiquette sera déplacée automatiquement de façon à rester toujours au même emplacement relatif.

Lors du calcul de la taille du rectangle représentant l'étiquette, il faudra tenir compte de la taille de la fonte utilisée.

Chevauchements d'éléments

Il se peut que certains éléments affichés se chevauchent. Pour traiter cela, et de façon à ce que le graphe soit le plus lisible possible, nous appliquons la règle suivante : les arcs sont d'abord affichés, suivis des étiquettes d'arcs, puis des sommets et enfin des étiquettes de sommets.

Eléments hors zone d'affichage

Dans notre conception, nous proposons d'afficher tous les éléments, même les éléments qui sont hors de la zone d'affichage ; nous pouvons nous le permettre car l'environnement de développement (Delphi) l'autorise sans générer d'erreurs.

Cela nous évite de devoir développer des fonctions de calcul compliquées pour savoir si un élément est affiché entièrement, partiellement ou pas du tout ; cela évite également de devoir développer les fonctions de *clipping* pour les éléments partiellement affichés. L'inconvénient est un gros problème d'optimisation lorsqu'on doit afficher un très grand graphe alors que la plupart de ses éléments ne sont pas affichables.

Notre justification est simple : les composants développés ici sont destinés à un usage didactique, et il n'y aura donc que très rarement de grands graphes qui pourraient pénaliser l'affichage.

Le degré de zoom dans l'affichage

Le degré de zoom est un des facteurs importants des calculs de l'affichage.

Tout d'abord, il modifie la taille de la zone d'affichage virtuelle : la longueur et la hauteur réelles sont divisées par ce facteur de zoom pour donner le nombre de pixels du plan qui seront affichés en largeur et en hauteur. Par exemple, une fenêtre qui fait 200 pixels de large sur 100 pixels de haut, avec un facteur de zoom de 2, affichera 100 pixels de large sur 50 pixels de haut du plan.

Les emplacements devront être calculés par rapport à cette zone d'affichage virtuelle. Prenons comme exemple un sommet affiché dans le plan à l'emplacement (30,30), alors que la fenêtre d'affichage fait 100 pixels de large sur 50 pixels de haut, que le degré de zoom est 2, et que le point de vue (rappelons qu'il correspond au point central de la fenêtre d'affichage) est aux coordonnées (50,25) : notre sommet sera affiché aux coordonnées (10,34)... La formule de calcul à utiliser pour calculer l'emplacement d'un point du plan réel dans le plan virtuel suite au degré de zoom est simple :

- Abscisse : $X_{nouveau}=(X*zoom)-((vueX-\frac{largeur}{2*zoom})*zoom)$
- Ordonnées : $Y_{nouveau}=(Y*zoom)-((vueY-\frac{hauteur}{2*zoom})*zoom)$

où $X_{nouveau}$ représente la nouvelle abscisse, $Y_{nouveau}$ représente la nouvelle ordonnée, zoom représente le degré actuel de zoom, vueX représente l'abscisse du point de vue, vueY représente l'ordonnée du point de vue, largeur représente la largeur réelle de la fenêtre en pixels, et hauteur représente la hauteur réelle de la fenêtre en pixels.

Ensuite, les diverses tailles des éléments sont multipliées par ce facteur : par exemple, un sommet qui a une taille de 30 pixels sera représenté, dans la zone d'affichage d'un composant ayant un facteur de zoom de 2, comme ayant une taille de 60 pixels. Cela est valable également pour l'épaisseur des arcs, les tailles des bordures des sommets.

6.5.3 L'implémentation

Implémentation de la « zone d'affichage »

Les activités d'analyse et de conception n'ont jamais évoqué le problème suivant : comment générer la zone d'affichage du composant V/M ?

Nous proposons la solution suivante, adaptée à notre environnement de développement : le composant V/M possède, par composition, un composant « PaintBox » qui prend toute la place occupée par le composant V/M, et offre toutes les fonctionnalités requises (dessin sur un canevas, capture d'événements système).

Affichage d'un graphe existant

Lorsqu'on souhaite afficher un graphe existant qui n'a pas d'informations d'habillage, il faut créer ces informations d'habillage et les remplir avec les informations par défaut. Le problème qui se pose concerne l'emplacement des sommets.

Nous proposons la règle suivante : les sommets seront placés le long d'un cercle, en commençant tout en haut de ce cercle. Chaque « côté » de la figure aura une longueur de 80 pixels (ce choix est arbitraire pour la lisibilité). Ainsi, un graphe de 3 sommets

sera représenté sous la forme d'un triangle, un graphe de 5 sommets sera représenté par un pentagone, etc. Le premier sommet sera arbitrairement placé aux coordonnées (0, 0). Une fois ces informations remplies, l'utilisateur n'aura plus qu'à disposer son graphe de façon à le rendre le plus lisible possible, via les opérations que nous verrons dans la cinquième itération, ou grâce aux méthodes de déplacement des sommets et des étiquettes.

Implémentation pour l'exigence d'extensibilité

La méthode d'affichage du graphe est implémentée de façon à appeler trois autres méthodes : dessin d'un sommet, dessin d'un arc et dessin d'une étiquette.

Ces trois nouvelles méthodes auxiliaires (privées) permettent ainsi au développeur de modifier les diverses formes de représentation selon ses besoins. La figure suivante résume notre propos :

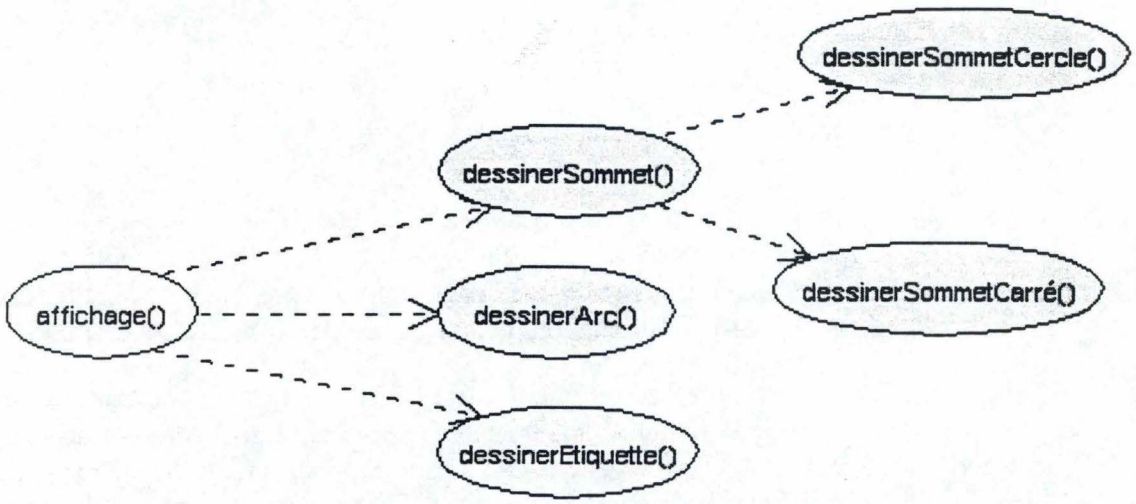


Diagramme 6-11 : Implémentation de la méthode « affichage() » par décomposition en plusieurs sous-méthodes, augmentant ainsi l'extensibilité du système

Par exemple, puisque nous proposons deux types de représentation de sommets, la méthode auxiliaire *dessinerSommet(IHSommet)* est constituée d'un appel d'une autre méthode auxiliaire selon le type de figure ; ainsi, si le développeur veut ajouter un nouveau type de représentation pour un sommet, il n'a plus qu'à créer une méthode permettant de dessiner le sommet selon cette représentation, et ajouter l'appel à cette méthode dans la méthode *dessinerSommet()* lorsque le sommet a son attribut *typeFigure...* qui correspond à cette nouvelle représentation.

Comme autre exemple, citons la facilité avec laquelle un développeur pourrait modifier la représentation d'arcs contraires : au lieu d'utiliser trois segments de droite, il pourrait calculer les arcs comme arcs d'ellipses.

Affichages des arcs

Un arc d'un sommet vers lui-même doit être représenté différemment ! Nous optons pour une représentation sous la forme d'un cercle, dont le point le plus bas (à 270 degrés par rapport à l'horizontale) correspond à l'emplacement du sommet, et de diamètre égal à la taille du sommet.

L'éventuelle indication de direction sera affichée au milieu de l'arc ; il faut tenir compte du fait que certains arcs sont affichés en 3 segments de droites, et dans ce cas afficher cet indicateur de direction au milieu du second segment de droite !

Méthodes auxiliaires

Quelques méthodes auxiliaires (privées) ont également été développées dans le composant V/M pour la facilité de développement :

- Calcul de l'angle d'un segment de droite déterminée par les coordonnées de ses deux extrémités ;
- Calcul de la longueur d'un segment de droite ;
- Calcul d'une coordonnée selon le degré de zoom, la taille de la fenêtre et le point de vue ;

Enfin, nous proposons également une méthode simple permettant de placer directement le point de vue à un emplacement précis, sans devoir utiliser la méthode de déplacement du point de vue. C'est une facilité non prévue dans les cas d'utilisation, mais qui ne coûte rien à développer, et pourra se révéler utile.

Les codes d'erreurs

Les erreurs suivantes sont définies dans cette itération :

Code erreur	Texte descriptif
24	Le sommet <1> ne possède pas d'étiquette.
25	L'arc de <1> à <2> ne possède pas d'étiquette.
26	Mauvais paramètre de zoom (<1>%).
27	Impossible de sélectionner le sommet <1>.
28	Impossible de sélectionner l'arc de <1> à <2>.
29	Impossible de sélectionner l'étiquette du sommet <1>.
30	Impossible de sélectionner l'étiquette de l'arc de <1> à <2>.

Tableau 6-5 : Description des erreurs générées par les opérations de cette itération n°4

Les définitions diverses

Les codes suivants sont définis lors de cette itération :

Elément	Codes	
Type d'élément sélectionné	• tsSommet = 1 • tsEtiquetteSommet = 2 • tsArc = 3 • tsEtiquetteArc = 4 • tsRien = 5	
Couleur (de fond, d'écriture, de bordure...)	• coNoir = 1 • coBlanc = 2 • coRouge = 3 • coVert = 4	• coBleu = 5 • coGris = 6 • coJaune = 7 • coOrange = 8

Tableau 6-6 : Description des codes définis pour cette itération n°4

6.5.4 Les tests

Les tests concernant cette itération sont réalisés via une petite application qui crée un graphe en mémoire, le garnit de 3 sommets et quelques arcs, puis l'affiche dans un composant V/M. Un bouton permet de lancer quelques sélections de sommets et d'arcs, de façon à vérifier que la visualisation fonctionne bien.

6.6 La cinquième itération

Rappelons que les cas d'utilisation qui sont construits lors de cette itération sont les suivants : CU033 (déplacement interactif du point de vue), CU034 (sélection interactive d'un élément du graphe), CU035 (déplacement d'éléments du graphe), CU036 (menu pop-up contextuel), CU037 (création interactive d'un sommet), CU038 (création interactive d'un arc), CU039 (création interactive d'une étiquette), CU040 (suppression d'éléments du graphe), CU041 (affichage d'une étiquette), CU042 (cacher une étiquette), CU043 (affichage de toutes les étiquettes), CU044 (cacher toutes les étiquettes), CU045 (modification interactive du degré de zoom).

6.6.1 L'analyse

L'analyse a été effectuée via des réalisations-analyses des cas d'utilisation concernés. Le lecteur trouvera ceux-ci à l'annexe 4.

6.6.2 La conception

On ne crée aucune nouvelle classe dans la conception de cette itération, mais on modifie des classes existantes.

Modification de la classe « Visualisateur / manipulateur »

La classe V/M possède de nouveaux attributs, de nouvelles méthodes, et capture 3 événements système. Le diagramme 6-12 indique la classe de conception complète.

Dans les nouveaux attributs, on distingue *ancienX* et *ancienY* qui n'ont aucune explication dans les réalisations-analyse de cas d'utilisation. Il s'agit simplement de l'emplacement du pointeur de la souris mis à jour à la fin de chaque déplacement de la souris ; cela sert à calculer le déplacement effectué à la souris lorsque l'événement capturé ne fournit que la nouvelle position du pointeur de la souris. On trouve également l'attribut *origineArcCréé* : lors de la création interactive d'un arc, il faut sauver le numéro interne du sommet d'origine avant que l'information concernant la sélection actuelle du composant V/M ne change.

Nous ne voyons pas de nouvelle méthode pour les opérations interactives de zoom et dézoom. En effet, nous allons appeler directement, lors du choix du point *ad hoc* du menu pop-up contextuel, la méthode déjà existante *modifierZoom()*. Il faudra fixer l'incrément ajouté ou enlevé lors de chaque appel de ces points de menu ; chaque développeur aura ses propres préférences ou exigences à remplir.

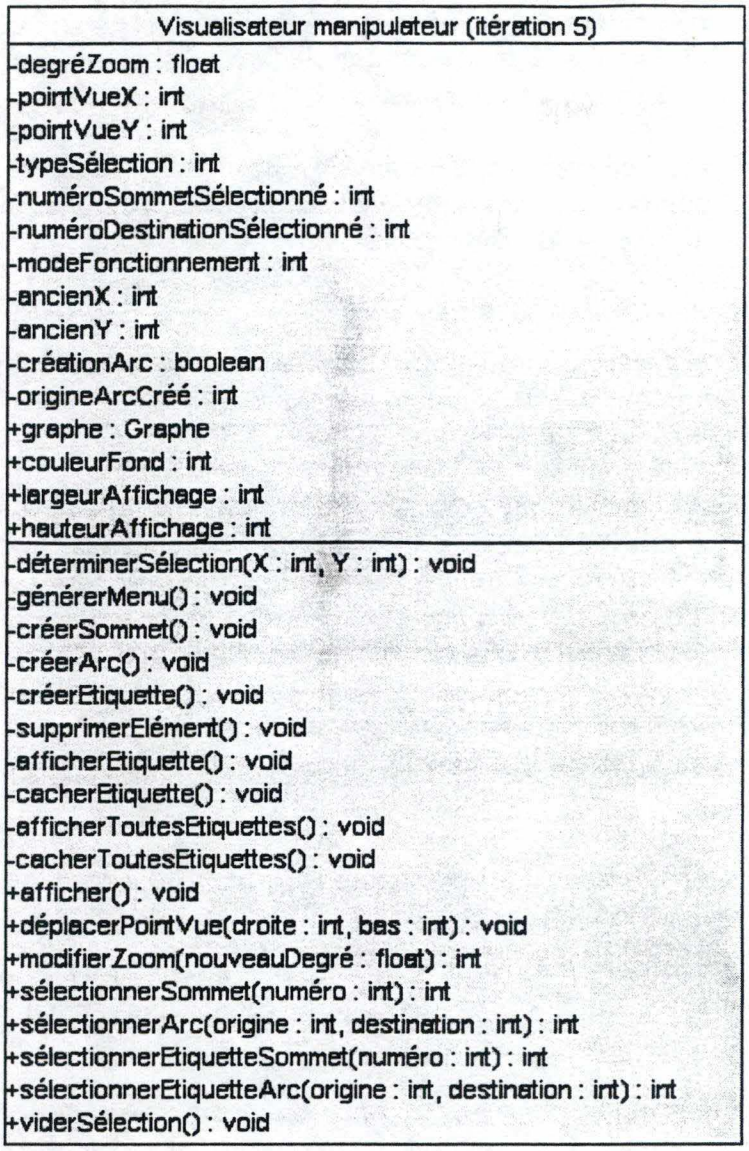


Diagramme 6-12 : Conception de la classe Visualisateur / manipulateur suite à l'itération n°5

Notons que le fonctionnement de la méthode *afficher()* est modifié de deux manières :

- lors de l'affichage des étiquettes, il faut maintenant vérifier si l'étiquette traitée doit être affichée ou pas (cette information se trouve dans les informations d'habillage de l'étiquette) ;

- si l'on est en mode de création d'arc, il faut afficher un arc temporaire entre son sommet d'origine et l'emplacement actuel du pointeur de la souris.

Pour remplir les fonctionnalités d'interactivité, il est nécessaire que cette classe capture les événements système suivants :

- boutonEnfoncé (X : int, Y : int, bouton : int) : cet événement système indique qu'on a enfoncé le bouton indiqué (représenté via un code) à l'emplacement (X, Y) de la zone d'affichage du composant ;

- boutonRelâché (X : int, Y : int, bouton : int) : cet événement système indique qu'on a relâché le bouton indiqué à l'emplacement (X, Y) de la zone d'affichage du composant ;

- mouvementSouris (X : int, Y : int) : cet événement système indique que le pointeur de la souris a été déplacée à l'emplacement (X, Y) de la zone d'affichage du composant.

Modification de la classe « Graphe »

La classe *Graphe* se voit ajouter deux nouvelles méthodes, que nous retrouvons dans la description de classe complète du diagramme 6-13 (voir page suivante) : *listeSommets()* et *listeArcs()*.

Les listes ordonnées renvoyées peuvent se contenter de n'avoir que les identifiants des éléments, et aucun contenu d'élément (*nil* généralement) ; en effet, on ne souhaite avoir que les numéros internes des sommets, ou les paires (origine, destination) des arcs du graphe. En parcourant ces listes de façon séquentielle, on est ainsi certain d'avoir accès à tous les éléments du graphe.

Modification de la classe « IHEtiquette »

La classe *IHEtiquette* possède un nouvel attribut : *affichée*, de type booléen. Il indique si l'étiquette doit être affichée ou non.

Analyse et conception des opérations suite aux événements système du composant V/M

Le traitement des événements systèmes par le composant V/M est assez délicat ; nous proposons une analyse pour aider le développeur à implémenter les comportements corrects.

Lorsque le bouton gauche est enfoncé, on modifie le mode de fonctionnement selon le type d'élément actuellement sélectionné :

- Sélection = vide ou arc : le mode de fonctionnement devient « déplacement du point de vue »

- Sélection = sommet, étiquette de sommet ou étiquette d'arc : le mode de fonctionnement devient « déplacement d'un élément »

Lorsque le bouton gauche est relâché, le mode de fonctionnement devient automatiquement « normal ».

Graphe (itération 5)
-orientation : boolean -numéroProchainSommet : int = 1 -représentationsInternes : ListeRI -erreurProduite : Erreur -infosHabillage : Informations d'habillage (arch) -sommetsPondérés : boolean -arcsPondérés : boolean
+ajouterSommet(numéro : int, poids : float, étiquette : String) : int +ajouterSommet(poids : float, étiquette : String) : int +modifierPoidsSommet(numéro : int, poids : float) : int +modifierEtiquetteSommet(numéro : int, étiquette : String) : int +supprimerSommet(numéro : int) : int +ajouterArc(origine : int, destination : int, poids : float, étiquette : String) : int +modifierPoidsArc(origine : int, destination : int, poids : float) : int +modifierEtiquetteArc(origine : int, destination : int, étiquette : String) : int +supprimerArc(origine : int, destination : int) : int +ajouterReprésentationInterne(codeRepresentation : int) : int +supprimerReprésentationInterne(codeRepresentation : int) : int +lireReprésentationInterne(codeRepresentation : int) : RepresentationInterne +erreur() : boolean +lireErreur() : Erreur +ordre() : int +nombreArcs() : int +estSymétrique() : boolean +estAsymétrique() : boolean +estAntisymétrique() : boolean +estComplet() : boolean +densité() : float +degréIntérieur(numéro : int) : int +degréExtérieur(numéro : int) : int +degré(numéro : int) : int +sauver(nomFichier : String) : int +listeSommets() : ListeLiéeOrdonnée +listeArcs() : ListeLiéeOrdonnée +lireSommet(numéro : int) : Sommet +lirePoidsSommet(numéro : int) : float +lireEtiquetteSommet(numéro : int) : String +lireArc(origine : int, destination : int) : Arc +lirePoidsArc(origine : int, destination : int) : float +lireEtiquetteArc(origine : int, destination : int) : String +lireIHSommet(numéro : int) : IH Sommet +lireIHArc(origine : int, destination : int) : IH Arc +lireIHEtiquette(numéro : int) : IH Etiquette +lireIHEtiquette(origine : int, destination : int) : IH Etiquette +lireIHSommetStandard() : IH Sommet +lireIHArcStandard() : IH Arc +lireIHEtiquetteStandard() : IH Etiquette +créerInformationsHabillage() : int +supprimerInformationsHabillage() : void

Diagramme 6-13 : Conception de la classe Graphe suite à l'itération n°5

Lorsque le bouton droit est enfoncé, on applique la table de décision suivante :

Test 1	Test 2	Action
Création arc = true	Sélection = sommet	Créer arc, passer en mode normal, création arc = false
	Sélection ≠ sommet	Création arc = false, passer en mode normal, générer menu
Création arc = false	Mode = normal	Générer menu
	Mode ≠ normal	(ne rien faire)

Tableau 6-7 : Table de décision à appliquer lors de l'événement « bouton droit de la souris enfoncé »

Lorsque l'on déplace la souris, on teste le mode de fonctionnement. Si ce mode est « normal », on détermine l'élément sélectionné au nouvel emplacement ; si ce mode est « déplacement du point de vue », on modifie le point de vue selon le déplacement détecté ; si ce mode est « déplacement d'un élément », on modifie l'emplacement de l'élément selon le déplacement détecté. Quoi qu'il arrive, on met à jour, à la fin de l'opération, l'emplacement actuel du pointeur de la souris (les attributs *ancienX* et *ancienY*) pour pouvoir traiter le prochain déplacement.

6.6.3 L'implémentation

Méthodes auxiliaires

Pour la facilité de programmation, nous avons implémenté les méthodes accessoires suivantes dans le composant « visualisateur / manipulateur » :

- *traductionCouleur (code : int) → Tcolor* : cette méthode fournit le code de couleur propre à Delphi correspondant au code de couleur utilisé dans l'application (défini au point 6.5.3).

- *coordX (X : int) → int* : cette méthode fournit la coordonnée (abscisse) réelle dans la zone d'affichage du composant V/M de l'abscisse d'un emplacement donné dans le plan, en tenant compte du point de vue et du degré de zoom.

- *coordY (Y : int) → int* : même chose que la méthode précédente, mais pour les ordonnées des emplacements.

- *proximité (Ax, Ay, Bx, By, X, Y, éloignement : int) → boolean* : cette méthode indique si l'emplacement (X,Y) se trouve à une distance maximale de (éloignement) pixels du segment de droite représenté par ses extrémités (Ax, Ay) et (Bx, By). Cette méthode privée est utilisée surtout lors de l'opération de détermination de la sélection suite à un déplacement de la souris.

- *calculLongueurDroite (Ax, Ay, Bx, By : int) → float* : cette méthode renvoie la longueur du segment de droite représenté par ses extrémités (Ax, Ay) et (Bx, By).

- *remplirPopupSommet, remplirPopupArc, remplirPopupEtiquette, remplirPopupGénéral, remplirPopupTous → void* : ces méthodes remplissent le menu pop-up contextuel selon le type d'élément sélectionné. La méthode *remplirPopupTous* est toujours appelée, et correspond aux options de menus toujours présents. Ces méthodes sont surtout utiles à une meilleur lisibilité du code.

Evénements supplémentaires

Nous proposons un certain nombre d'événements supplémentaires générés par les activités de cette itération :

- Sélection d'un sommet / dé-sélection d'un sommet : cet événement a comme paramètre le numéro du sommet (dé-)sélectionné.

- Sélection d'un arc / dé-sélection d'un arc :cet événement a comme paramètre les numéros internes des sommets d'origine et de destination de l'arc (dé-)sélectionné.

- Sélection d'une étiquette de sommet / dé-sélection d'une étiquette de sommet : fournit le numéro du sommet de l'étiquette sélectionnée comme paramètre.

- Sélection d'une étiquette d'arc / dé-sélection d'une étiquette d'arc : fournir les numéros des sommets d'origine et de destination de l'arc de l'étiquette sélectionnée.

- Clic sur sommet : indique que l'utilisateur a cliqué (bouton gauche relâché) sur un sommet donné, en fournissant le numéro interne de ce sommet.

- Clic sur un arc : indique que l'utilisateur a cliqué sur un arc donné, en fournissant les numéros internes des sommets d'origine et de destination de l'arc.

- Clic sur une étiquette de sommet : indique que l'utilisateur a cliqué sur l'étiquette d'un sommet donné, en fournissant le numéro interne de ce sommet.

- Clic sur une étiquette d'arc : indique que l'utilisateur a cliqué sur l'étiquette d'un arc donné, en fournissant les numéros internes des sommets d'origine et de destination de cet arc.

- Déplacement d'un sommet : indique que l'utilisateur vient de déplacer un sommet, en fournissant le numéro interne de ce sommet et son nouvel emplacement dans le plan.

- Fin du déplacement d'un sommet : indique que l'utilisateur vient de terminer le déplacement d'un sommet (l'utilisateur a relâché le bouton de la souris), en fournissant son numéro interne et son emplacement actuel.

- Déplacement d'une étiquette de sommet : indique que l'utilisateur vient de déplacer une étiquette de sommet, en fournissant le numéro interne du sommet et le nouvel emplacement de l'étiquette dans le plan.

- Fin du déplacement d'une étiquette de sommet : indique que l'utilisateur vient de terminer le déplacement d'une étiquette de sommet, en fournissant le numéro interne du sommet et l'emplacement actuel de l'étiquette.

- Déplacement d'une étiquette d'arc : indique que l'utilisateur vient de déplacer une étiquette d'arc, en fournissant les numéros internes des sommets d'origine et de destination de l'arc et le nouvel emplacement de l'étiquette dans le plan.

- Fin du déplacement d'une étiquette d'arc : indique que l'utilisateur vient de terminer le déplacement d'une étiquette d'arc, en fournissant les numéros internes des sommets d'origine et de destination de l'arc et l'emplacement actuel de l'étiquette.

Erreurs générées

Le tableau 6-8 résume les erreurs générées par les opérations de cette itération :

Code erreur	Texte descriptif
31	Impossible de créer un nouveau sommet.
32	Impossible de créer un nouvel arc.
33	Impossible de créer une nouvelle étiquette.

Tableau 6-8 : Description des erreurs générées par les opérations de cette itération n°5

Les définitions diverses

Les codes suivants sont définis lors de cette itération :

Elément	Codes
modeFonctionnement	• mfNormal = 1 • mfDéplacementPointVue = 2 • mfDéplacementElément = 3

Tableau 6-9 : Description des codes définis pour cette itération n°5

Limites du facteur de zoom

Comme indiqué dans la conception, il faut définir les limites autorisées du degré de zoom lors de l'implémentation. Dans notre cas, nous autorisons les valeurs dans l'intervalle] 0, 10], ce qui nous semble largement suffisant.

6.6.4 Les tests

Les tests pour les opérations de cette itération sont effectués grâce à l'application de visualisation/manipulation que nous verrons au chapitre suivant (voir 7.3.3). On peut ainsi vérifier que toutes les opérations de manipulation sont bien effectuées, ainsi que la visualisation du fonctionnement d'un algorithme.

6.7 Déploiement du système

Le diagramme 6-14 indique la manière dont nous avons divisé le système et ses éléments en divers paquetages de compilation, ainsi que les relations qu'ils entretiennent entre eux.

On voit que le paquetage « divers » est utilisé par les autres paquetages ; il contient les éléments divers utilisables partout, et est donc un paquetage de service.

Les graphes sont divisés en trois paquetages : le premier contient les fonctionnalités des graphes en tant que classe frontière (c'est-à-dire dialoguant avec l'utilisateur dans un programme), le second s'occupe des représentations internes, et le dernier s'occupe des informations d'habillage.

Les algorithmes ont leur propre paquetage, ainsi que le visualisateur.

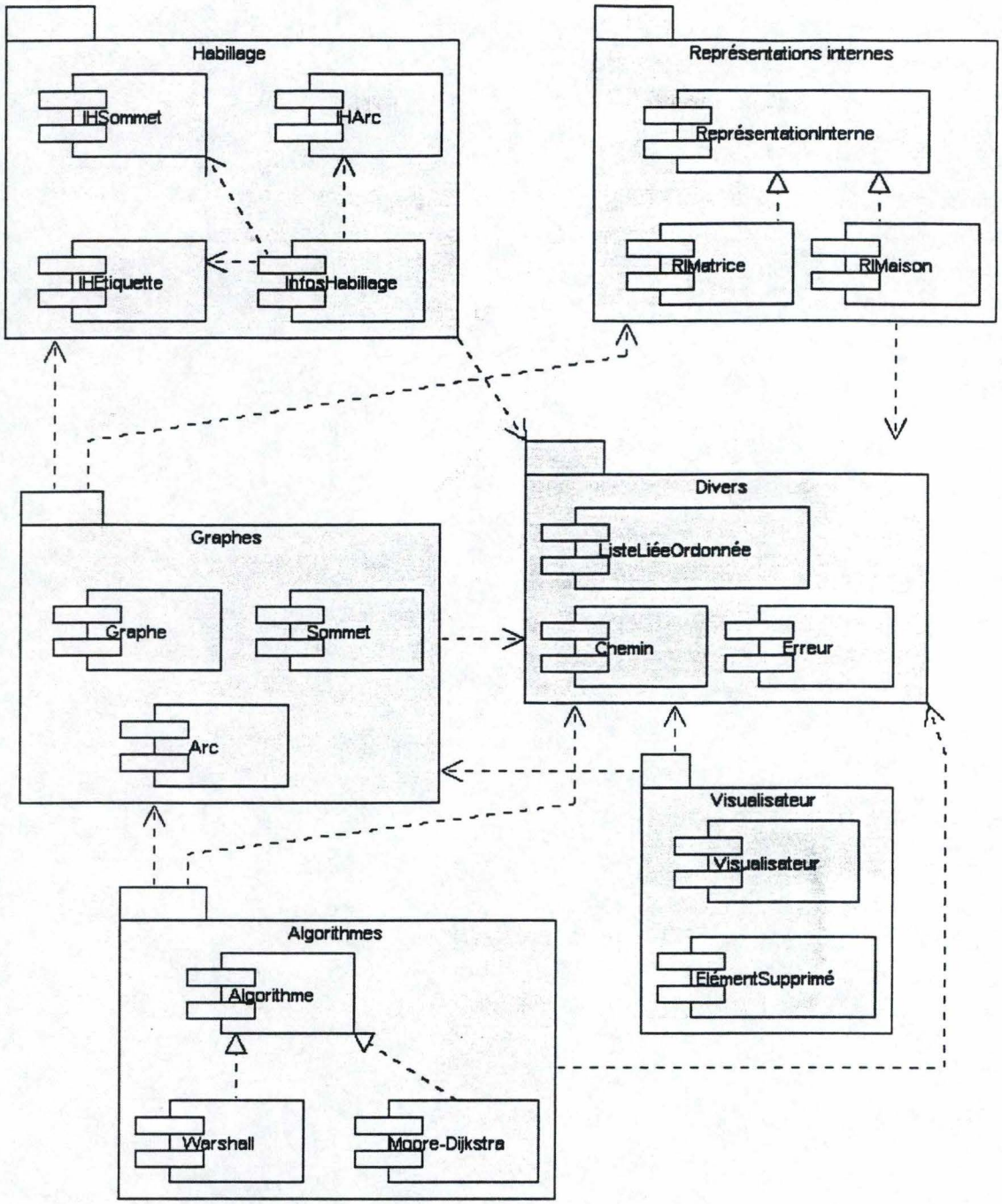


Diagramme 6-14 : Déploiement du système en divers paquetages contenant chacun divers composants clés du système

7

La phase de transition

La démarche méthodologique – Modifications demandées – Création de programmes exemples – Revue post mortem

Le chapitre 7 traite de la phase de transition : elle correspond à la mise en service d'une version *beta* du système, à la correction des anomalies rencontrées par les utilisateurs finaux du système, et à la mise en exploitation du système fini.

Nous y verrons les améliorations demandées à ce point, et l'utilisation qu'on a faite du système fini.

7.1 La démarche méthodologique

Cette phase est relativement importante pour des projets de grande envergure ; pour notre propos, elle est évidemment fort allégée.

7.1.1 Version beta et retouches

La première activité importante de cette phase est la fourniture d'une version *beta* du système aux utilisateurs. Ceux-ci devront effectuer les tests qu'ils jugeront nécessaires, afin de découvrir les anomalies, défaillances, manquements du système.

On doit toujours s'attendre à des retouches du système : le produit parfait n'existe pas, même si la méthodologie essaie d'atteindre la meilleure qualité possible. De plus, l'utilisation d'une méthode itérative et incrémentale nous habitue aux retouches.

7.1.2 Finalisation des artefacts

Quand le système a été retouché et correspond aux attentes de ce cycle de vie, il est alors temps de finaliser les divers artefacts construits : code, modèles d'analyse et de conception, plan de déploiement, modes d'emploi, etc.

7.1.3 Revue post-mortem du projet

Une partie importante de cette phase consiste en la revue *post mortem* du projet : il faut analyser les calendriers initial et réel, les coûts budgétés et effectifs, les objectifs qui ont été atteints ou pas et pourquoi, etc. Le but principal est de se créer des archives et des métriques permettant une meilleure gestion ultérieure grâce à l'expérience acquise.

7.2 Modifications demandées

Après démonstration du système, il est apparu à l'utilisateur final (et sponsor du projet), M. Leclercq, qu'il manquait une fonctionnalité d'annulation de la dernière opération effectuée via le visualisateur / manipulateur.

Une rapide analyse et estimation du temps de développement a conclu à l'implémentation de ce nouveau besoin dans ce cycle de vie.

7.2.1 Le cas d'utilisation

Pré-condition

L'utilisateur emploie un composant V/M dans une application, qui affiche un graphe garni. Il souhaite annuler la dernière opération qu'il vient d'effectuer.

Flot d'événements

1. Le cas d'utilisation est invoqué lorsque l'utilisateur sélectionne, dans le menu pop-up contextuel du composant V/M, l'option permettant d'annuler la dernière suppression qu'il a effectuée.

2. Le système recrée les éléments supprimés dans leur état d'avant suppression.

3. Le système vide les informations concernant la dernière suppression annulable.

4. Le cas d'utilisation se termine par l'affichage du graphe.

Chemins de rechange

Dans l'étape 1, le point de menu n'apparaît dans le menu pop-up contextuel que s'il y a une suppression à annuler ; il ne faut donc pas tester s'il y en a bien une.

Dans l'étape 2, il se peut qu'il y ait plusieurs éléments à recréer suite aux effacements en cascade : un arc supprimé peut avoir entraîné des arcs avec lui, ou une étiquette, par exemple.

Notes – exigences supplémentaires

Le cas d'utilisation ne s'occupe que de l'annulation des suppressions. En effet, un déplacement d'élément peut toujours être annulé à la main, et un élément ajouté peut toujours être supprimé facilement. Cette limitation du *scope* du cas d'utilisation le rend plus facile à analyser et implémenter, et nous autorise à le prévoir dans ce cycle de vie.

7.2.2 L'analyse du cas d'utilisation

Le diagramme 7-1 indique les collaborations du cas d'utilisation :

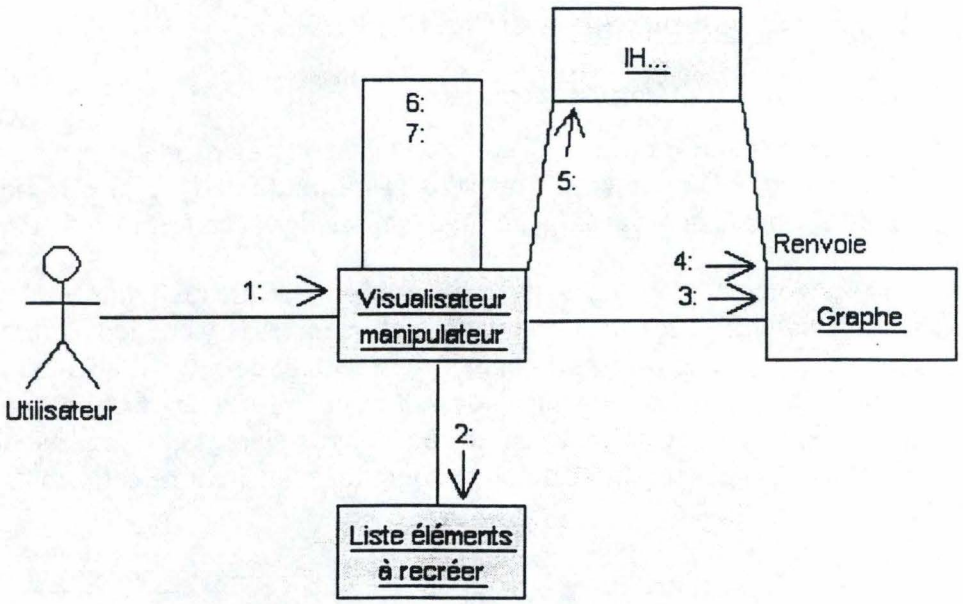


Figure 7-1 : Diagramme de collaboration du nouveau cas d'utilisation

Les messages échangés sont :

1. Sélection dans un menu pop-up de l'option permettant d'annuler la dernière suppression dans le graphe.
2. * Lire un élément supprimé à recréer dans la liste des éléments supprimés.
3. Créer sommet (numéro, poids, étiquette) ; Créer arc (origine, destination, poids, étiquette).
4. Fournir les informations d'habillage de l'élément recréé.
5. Modifier directement les informations d'habillage de l'élément recréé.
6. Vider la liste des éléments supprimés à recréer.
7. Affichage.

Analyse

Une nouvelle responsabilité pour la classe V/M : annuler la dernière suppression.

On constate qu'il faut créer une nouvelle classe contenant les informations des éléments supprimés lors de la dernière suppression. Cette classe peut contenir plusieurs éléments différents. La logique nous indique qu'on pourra avoir simultanément les informations d'un sommet (y compris ses infos d'habillage), les informations d'habillage d'une étiquette de sommet, et une liste d'informations d'arcs (y compris leurs infos d'habillage) et les informations d'habillage de leurs étiquettes.

Cette classe devra contenir des copies des données des éléments supprimés, afin de les recréer dans leur état complet d'avant suppression.

7.2.3 La conception du nouveau cas d'utilisation

Nouvelle classe : Eléments supprimés

Nous créons une nouvelle classe contenant les éléments supprimés. Pour plus de facilité, nous allons la concevoir sous la forme d'une liste liée ordonnée d'éléments, en réutilisant ainsi la classe générique réalisée dès la conception de l'architecture.

Chaque élément de cette liste sera numéroté séquentiellement. Les éléments eux-mêmes seront représentés par des objets simples, ayant deux attributs publics : le premier attribut est un code entier représentant le type d'élément à recréer, et le second attribut est l'élément à recréer lui-même. Pour contenir tous les différents types d'éléments, ce second attribut sera de type *Object*, et il faudra donc effectuer des *downcast* lors de leur utilisation. Le diagramme 7-1 indique cette conception :

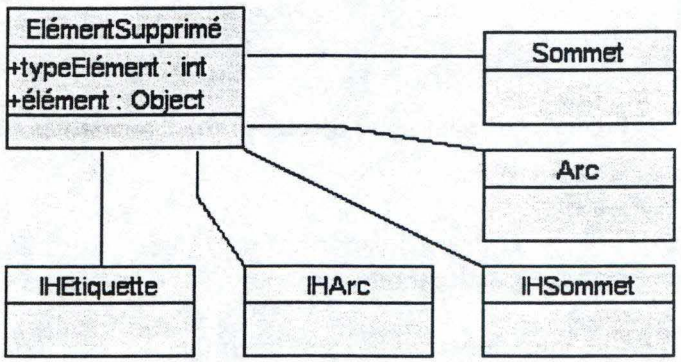


Diagramme 7-1 : Conception de l'élément de la liste liée des éléments supprimés à recréer

Les données contenues dans les objets de l'attribut *élément* sont des copies faites des données réelles avant leur suppression.

Il faudra être très attentif à l'ordre dans lequel on recrée les éléments : par exemple, il est impossible de recréer un arc avant d'avoir recréé le sommet représentant une de ses extrémités. Nous proposons l'ordre suivant : le sommet, les infos d'habillage du sommet, les infos d'habillage de l'éventuelle étiquette du sommet, un arc, les infos d'habillage de cet arc, les infos d'habillage de l'éventuelle étiquette de l'arc.

Cela nous amène à un autre problème : si l'élément supprimé est une étiquette seule, comment faire pour la représenter dans cette liste ? Nous proposons donc de dériver deux classes à partir de la classe *IHEtiquette* (diagramme 7-2) :

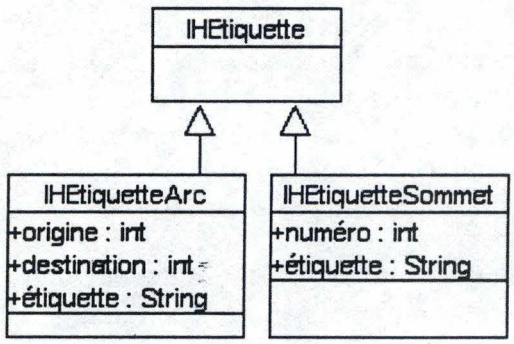


Diagramme 7-2 : Deux classes dérivées de la classe IHEtiquette

Ainsi, nous avons toutes les informations nécessaires pour la bonne marche de l'opération.

Adaptation de la classe Visualisateur / manipulateur

Cette classe possède un nouvel **attribut** privé : *élémentsSupprimés*, de type *ListeLiéeOrdonnée*.

Elle possède également une nouvelle **méthode** publique : *annulerSuppression() → void*.

Lors de sa création, cette classe initialisera le nouvel attribut à *nil*. Les méthodes de génération du menu pop-up contextuel seront adaptées de façon à proposer l'option d'annulation si l'attribut *élémentsSupprimés* n'est pas vide.

7.2.4 L'implémentation

Pour éviter la majorité des problèmes, nous avons implémenté cette nouvelle fonctionnalité de façon à ce que toute création interactive vide la liste des éléments supprimés, et empêche donc la fonctionnalité.

Erreurs générées

Le tableau 7-1 résume les erreurs générées par cette nouvelle opération :

Code erreur	Texte descriptif
34	Impossible de recréer tous les éléments supprimés du graphe.

Tableau 7-1 : Description des erreurs générées par cette nouvelle opération

Les définitions diverses

Les codes suivants sont définis pour cette opération :

Elément	Codes
Type d'élément supprimé	• teSommet = 1 • teArc = 2 • telHSommet = 3 • telHArc = 4 • telHEtiquetteSommet = 5 • telHEtiquetteArc = 6

Tableau 7-2 : Description des codes définis pour cette nouvelle opération

7.3 Création de programmes exemples

L'un des objectifs du projet (voir le chapitre 1) était la réalisation d'applications utilisant les composants conçus et réalisés. Ces applications doivent démontrer le bon fonctionnement du système, sa facilité d'emploi et sa puissance.

Etant donné que nous avons réalisé deux algorithmes (Warshall et Moore-Dijkstra), nous avons repris les exemples du cours de M. Leclercq pour en faire des applications de démonstration. Nous avons également réalisé une petite application de démonstration du fonctionnement du composant V/M.

7.3.1 Démonstration de l'algorithme de Warshall

Cette simple application va refaire le fonctionnement de l'algorithme de Warshall vu dans l'exemple du cours de M. Leclercq.

On démontre ici : le bon fonctionnement d'un objet Graphe (garnissage programmé), le bon fonctionnement de l'algorithme Warshall, la capture des informations de fonctionnement de cet algorithme (exécution pas à pas) et la façon dont on peut les employer dans un but didactique (affichage à l'écran dans ce cas).

Interface utilisateur

L'interface utilisateur est volontairement simple : l'application contient deux boutons permettant soit de lancer l'exécution de l'algorithme, soit de quitter l'application. Les informations issues de l'exécution de l'algorithme seront affichées dans une zone texte, représentée par un composant standard Delphi appelé TMemo : ce composant permet d'afficher autant de lignes de texte que désiré, offre des barres de défilement vertical et/ou horizontal, et peut recevoir dynamiquement de nouvelles lignes de texte.

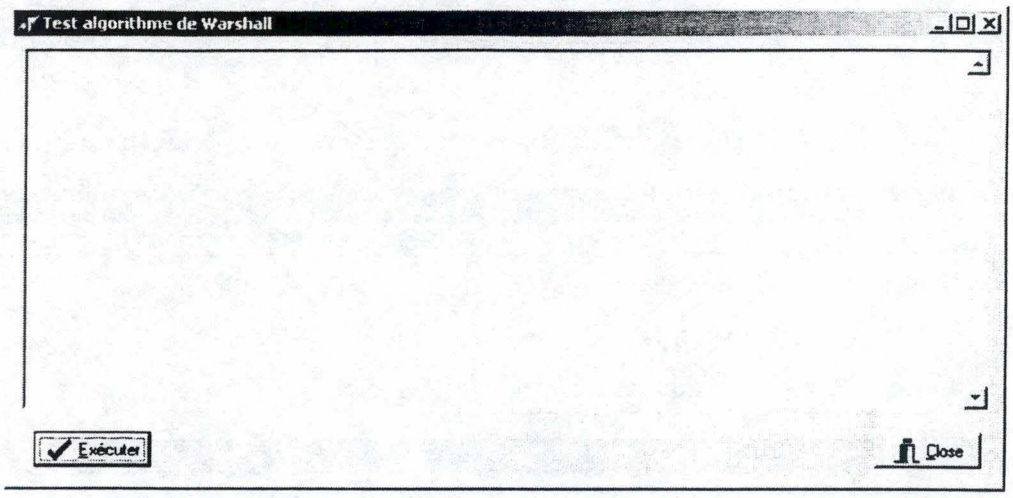


Figure 7-2 : Interface utilisateur de l'application « test Warshall »

Conception de l'application

L'application contient :

- un composant Graphe (orienté), qu'on garnira au début de l'exécution de l'application ;
- un composant Algorithme Warshall, auquel on assignera le composant Graphe quand celui-ci aura été garni, et qu'on exécutera pas à pas.
- un composant Tmemo qui affichera les données de fonctionnement de l'algorithme Warshall au fur et à mesure qu'on les capture.

Implémentation

L'application est très simplement implémentée. On commence par définir ce qui se passe lorsque l'utilisateur clique sur le bouton « exécuter » : on garnit le graphe avec

les données prévues (la figure 7-3 montre ce graphe visuellement), on assigne ce graphe à l'instance de l'algorithme de Warshall, et on lance l'exécution de celui-ci.

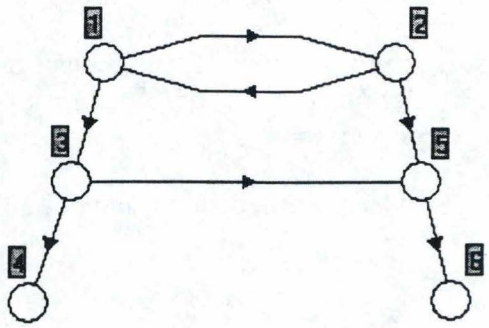


Figure 7-3 : Apparence visuelle du graphe garni pour l'exécution du test de l'algorithme de Warshall.

Ensuite, on implémente le traitement à effectuer lors de l'événement « boucle dans l'exécution » de cet algorithme. L'événement est accompagné des données de fonctionnement, sous la forme d'un objet de type *Object* ; on en tire la valeur de la variable K (après un downcast vers le type de données correct TdataAlgWarshall) qu'on affiche, et la matrice qu'on affiche en la parcourant ligne par ligne. Les affichages se font en ajoutant des lignes au composant TMemo.

Pour une vision plus didactique, nous capturons également l'événement proposé par notre implémentation spécifique de l'algorithme de Warshall : examen d'une case de la matrice ; à chaque fois que cet événement se produit, nous affichons dans le champs mémo les coordonnées de cette case.

Exemple de fonctionnement

La figure 7-4 montre une capture d'écran de ce qui est affiché pendant l'exécution de l'application.

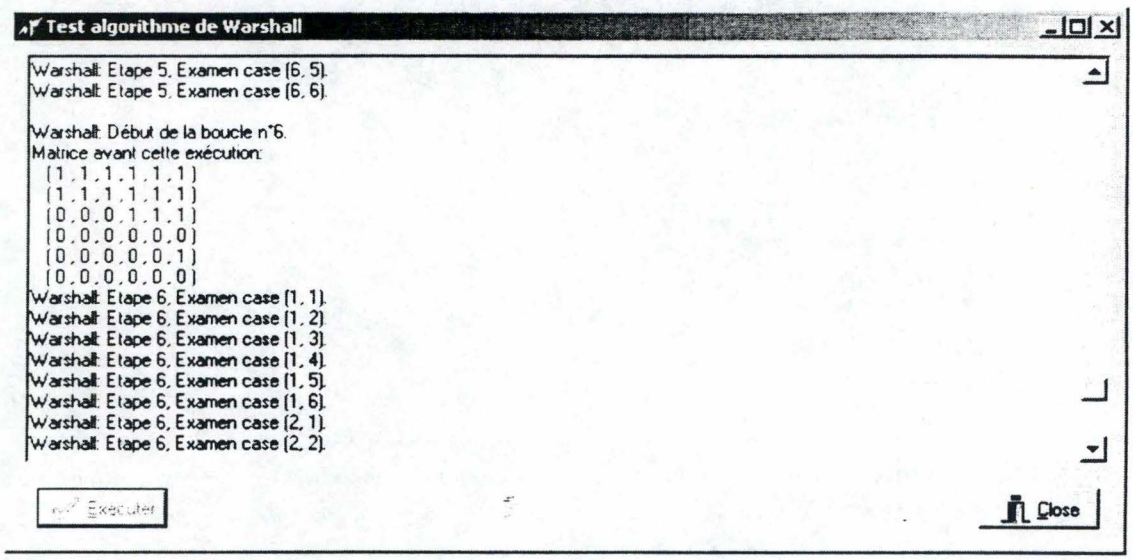


Figure 7-4 : Capture d'écran du fonctionnement de l'application de test de l'algorithme Warshall

7.3.2 Démonstration de l'algorithme de Moore-Dijkstra

Cette simple application va refaire le fonctionnement de l'algorithme de Moore-Dijkstra vu dans l'exemple du cours de M. Leclercq.

L'application suit en gros le même cheminement que l'application vue dans la section précédente (7.3.1 Démonstration de l'algorithme de Warshall) : même interface utilisateur, même conception.

Le graphe qu'on utilise est garni différemment. La figure 7-5 montre ce graphe :

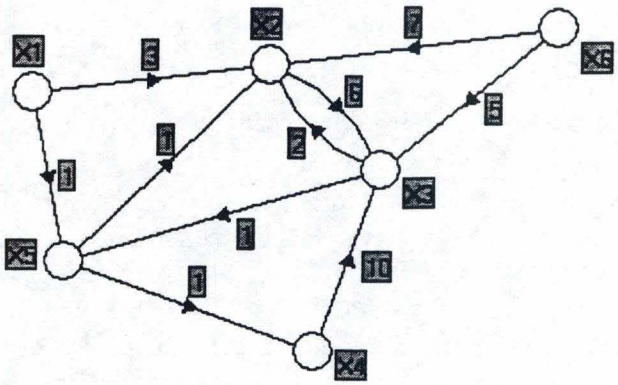


Figure 7-5 : Graphe utilisé dans l'application de test de l'algorithme de Moore-Dijkstra

Pendant l'exécution de l'application, on capture divers éléments d' l'algorithme : le début de son exécution (uniquement à titre de démonstration), les diverses boucles de son exécution (on affiche les poids et étiquettes de chaque sommet), les choix du sommet Y à traiter (on affiche le numéro du sommet choisi et son poids), et les diverses modifications apportées aux chemins minimaux (on affiche le sommet modifié, ses anciens et nouveaux poids et étiquettes).

La figure 7-6 est une capture d'écran du fonctionnement de cette application :

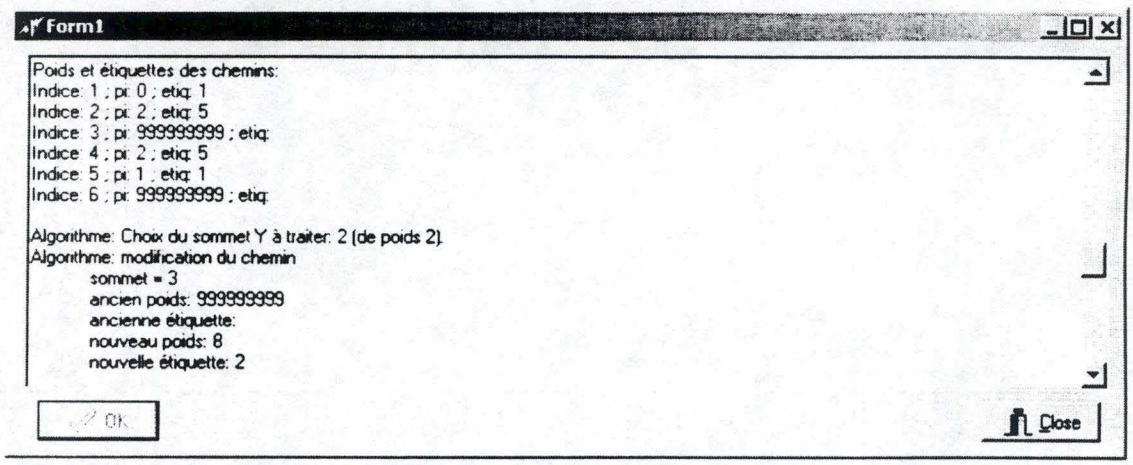


Figure 7-6 : Capture d'écran du fonctionnement de l'application de test de l'algorithme de Moore-Dijkstra

7.3.3 Démonstration du visualisateur / manipulateur

Cette petite application propose un composant V/M utilisable de façon didactique.

On démontre ici : le bon fonctionnement d'un objet Graphe (garni interactivement), le bon fonctionnement du composant V/M, l'exécution visuelle des algorithmes.

Interface utilisateur

L'interface utilisateur est ici aussi fort simple. Outre le composant V/M, on propose un menu permettant les options suivantes : créer un nouveau graphe, lire un graphe depuis un fichier, écrire le graphe affiché dans un fichier, quitter l'application, exécuter l'algorithme de Warshall, exécuter l'algorithme de Moore-Dijkstra.

Nous affichons également un champs mémo permettant d'ajouter diverses informations textuelles sur les opérations. Il se trouve sous le composant V/M.

Conception

L'application contient :

- un composant Graphe (orienté) qui sera garni interactivement par l'utilisateur grâce au composant V/M ;
- un composant Algorithme Warshall, auquel on assignera le composant Graphe quand celui-ci aura été garni, et qu'on exécutera pas à pas à la demande de l'utilisateur ;
- un composant Algorithme Moore-Dijkstra, auquel on assignera le composant Graphe quand celui-ci aura été garni, et qu'on exécutera pas à pas à la demande de l'utilisateur ;
- un composant Tmemo qui affichera les données de fonctionnement des algorithmes au fur et à mesure qu'on les capture, ainsi que divers messages systèmes (lecture d'un fichier réussie ou pas, etc.) ;
- un composant Visualisateur / manipulateur, auquel on assigne le graphe.

Implémentation

Les opérations de garnissage du graphe sont totalement interactives et suivent les standards prévus par le composant VM.

Pour exécuter un algorithme Warshall, l'utilisateur n'a qu'à utiliser le point de menu ad hoc. Ce point de menu ne fait qu'assigner le graphe à l'instance de l'algorithme, et exécuter l'algorithme pas à pas, en appelant sa méthode *executerPasAPas()* ; l'affichage visuel des modifications se fait automatiquement.

Pour exécuter un algorithme Moore-Dijkstra, il faut qu'on connaisse la racine de départ de l'algorithme ; dans ce cas, on demande donc à l'utilisateur de cliquer d'abord sur la racine voulue. On capture cet événement (généré par le composant V/M) et on exécute alors l'algorithme. Vu qu'il est peu visuel, on affiche les informations de traitement dans le champs mémo, comme pour l'application de test de cet algorithme vue au point 7.3.2.

On remarque également que cet algorithme fonctionne en se basant sur les poids des arcs ; or, le garnissage interactif ne change pas ces poids ! On démontre ici la

puissance de notre outil : grâce aux événements « clic sur sommet » et « clic sur arc », on affiche une fenêtre modale d'encodage du poids de l'élément sur lequel l'utilisateur a cliqué, et on modifie ainsi les poids des éléments du graphe. Comme il y a collision d'utilisation du « clic sur sommet » entre cet encodage du poids et la désignation de la racine pour l'algorithme de Moore-Dijkstra, on utilise un simple drapeau pour savoir comment traiter un clic sur un sommet.

Le sauvetage et la lecture des graphes sont également implémentés dans cette application, permettant ainsi à l'utilisateur de préparer ses graphes avant démonstration du fonctionnement des algorithmes (toujours notre but didactique !)

Ci-dessous, on trouvera quelques captures d'écran de cette application :

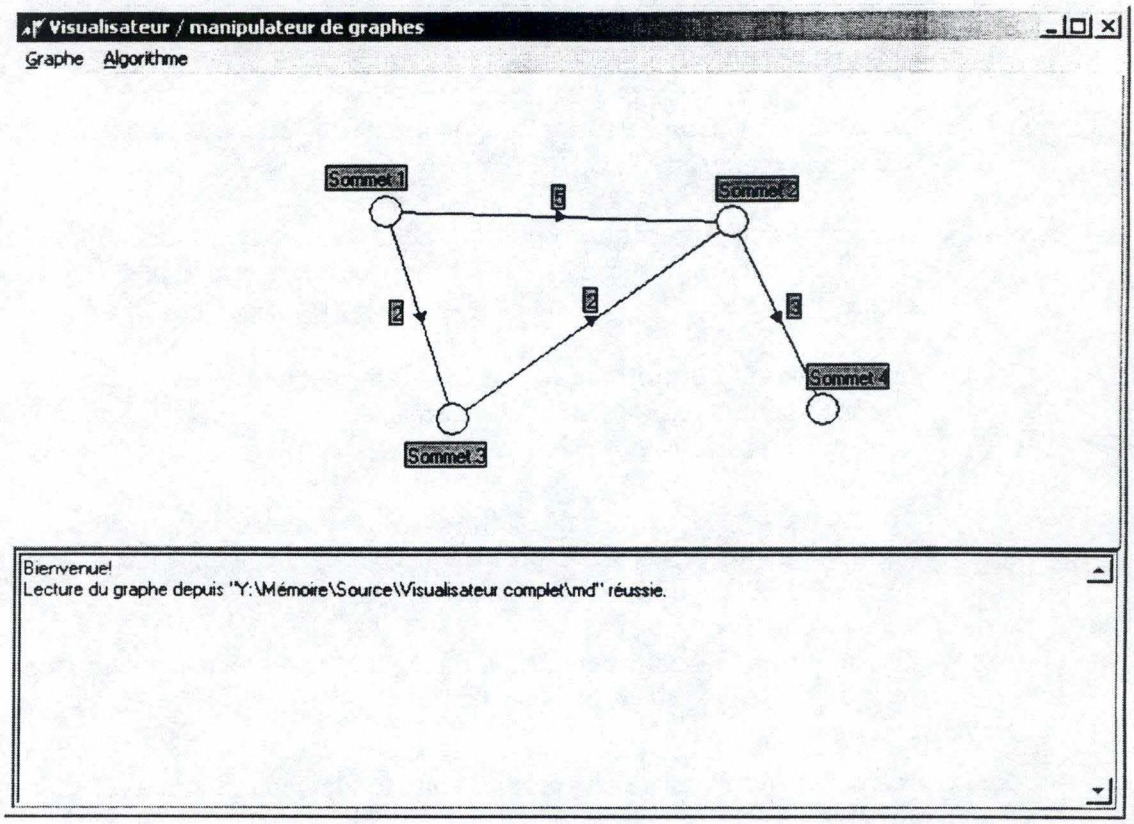


Figure 7-7 : un graphe a été lu depuis un fichier, et il est affiché dans le V/M.

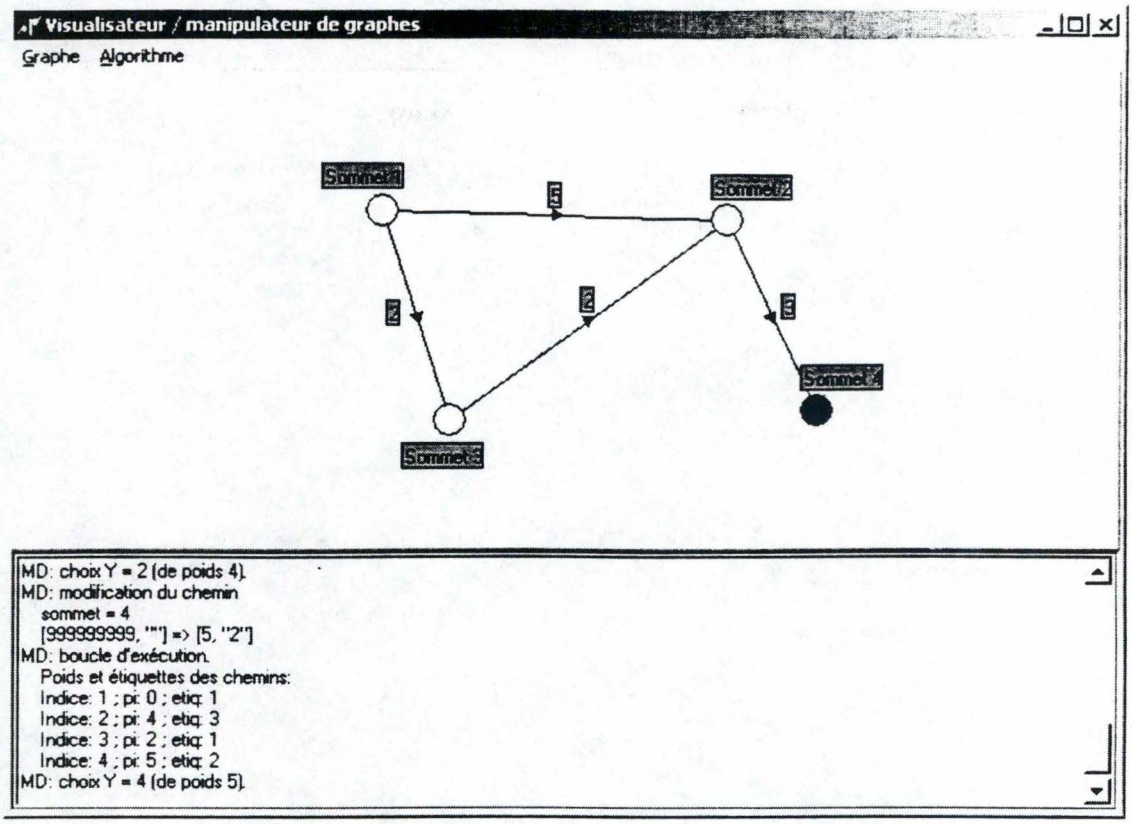


Figure 7-8 : Affichage visuel et textuel du fonctionnement de l'algorithme Moore-Dijkstra

7.4 Revue post mortem

Joliment intitulée « revue post-implémentation » par Robert Buttrick [Buttrick R., 2000], cette opération est importante afin de retirer l'expérience acquise dans le projet.

7.4.1 Délais

Le délai initialement prévu de deux mois/homme de développement est passé en réalité à un peu plus de trois mois/homme. Cependant, nous estimons que la grande majorité du dépassement est dû au fait que le développement s'est fait par petites phases, suite aux contraintes de vie professionnelle et familiale de l'auteur.

7.4.2 Expérience

La plus grande expérience retenue du projet est la facilité d'utilisation de la méthodologie ; les enchaînements d'activités, de phases et d'itérations sont assez naturels. Nous avons également apprécié la traçabilité entre cas d'utilisation, réalisation-analyse de cas d'utilisation et éléments de conception.

7.4.3 Gestion et administration

Malheureusement, le projet est de trop petite envergure pour nécessiter des opérations de gestion de projet. Cela nous aurait été très utile professionnellement.

8

Conclusions

8.1 Récapitulation de notre travail

8.1.1 Les objectifs du projet

Les quatre objectifs du projet ont été atteints.

Nous avons conçu et spécifié les objets indispensables à la création, la manipulation et la visualisation de graphes, ainsi qu'à l'utilisation d'algorithmes sur ces graphes. L'exigence non fonctionnelle d'extensibilité a également été respectée : l'architecture de notre système est suffisamment ouverte pour permettre l'ajout de nouveaux algorithmes, de nouveaux types de représentations internes, ainsi que de nouvelles façons d'afficher les éléments de graphes.

Le système que nous proposons a été réalisé en Delphi, et offre des composants utilisables directement dans cet environnement de développement.

Enfin, trois applications à caractère didactique ont été réalisées à titre de démonstration de la validité du système.

8.1.2 La méthodologie

La méthodologie utilisée, basée sur le « Processus Unifié », a été un soutien appréciable pour notre travail. Elle a évidemment nécessité quelques adaptations, principalement des allègements de points inutiles dans un projet de si petite envergure.

Nous espérons avoir démontré l'intérêt d'une telle méthode itérative et incrémentale, par le développement d'éléments au fur et à mesure de l'avancement du projet, en se basant sur une architecture de référence : le produit arrive plus rapidement à un état fonctionnel (quoique incomplet), les changements sont plus facilement intégrés au processus, les anomalies sont plus vite découvertes.

La décomposition en phases distinctes, elles-mêmes décomposées en itérations, a été bien montrée par l'utilisation de chapitres séparés dans ce travail. Nous espérons également avoir bien fait comprendre le fait que chaque itération peut voir les cinq enchaînements d'activités (besoins, analyse, conception, implémentation, test) à des degrés divers.

8.1.3 Notre apport au domaine

Notre travail n'est certainement pas une nouveauté ; il suffit de voir le chapitre 3 (l'état de l'art) pour se convaincre que d'autres travaux sur ce sujet ont été développés précédemment.

Notre apport original a été :

- une abstraction des spécifications par rapport au langage de développement (la plupart des produits découverts fonctionnent en Java) ;
- la possibilité d'employer plusieurs types de représentations internes, simultanément au besoin ;
- la possibilité de visualiser le fonctionnement d'algorithmes pendant leur exécution.

8.1.4 Nos regrets

Malgré le fait que notre produit fonctionne selon les besoins exprimés, nous regrettons cependant son manque d'optimisation. Il est évident que ce faible niveau d'optimisation est lié aux exigences de généricité et d'extensibilité, ainsi qu'à l'abstraction par rapport aux langages de développement ; il devrait être possible d'améliorer cette caractéristique du système, mais cela est hors du *scope* de notre travail.

8.1.5 Comparaison par rapport à l'état de l'art

Il est intéressant de comparer notre système avec ceux vus dans le chapitre 3 (état de l'art). Voici notre estimation :

Produit	GRIN	aiSee	JDSL	DGE	JViews	VGJ	OpenJGraph	Notre système
Graphe	+	--	+	+	--	+	+	+
Repr. Internes	--	?	=	--	?	--	=	++
Algorithmes	+	?	+	--	?	--	+	+
Visualisation	++	++	--	=	++	+	+	+
Manipulation	=	--	--	+	--	+	+	+
Extensibilité	?	?	++	?	?	--	+	+

Tableau 8-1 : comparaison de notre système avec ceux découverts lors de l'analyse de l'état de l'art. Légende : « ++ » signifie que le produit est excellent dans ce domaine, « + » signifie un bon produit, « = » signifie un produit moyen, « - » signifie que le produit est faible, et « -- » signifie que le produit est mauvais (ou n'offre pas de possibilités dans ce domaine).

8.2 Perspectives et améliorations

Notre système est fonctionnel dans son état actuel. Cependant, il souffre encore de quelques manques par rapport à d'autres produits existants ; il est à noter que ces manques ne sont pas dus à un mauvais travail, mais simplement au *scope* du projet qui ne les incluait pas.

8.2.1 Possibilités de réaffichage de graphes

A notre sens, le manque le plus flagrant de notre système est l'impossibilité actuelle d'appliquer des algorithmes de réaffichage (*layout*) aux graphes existants. L'utilisateur doit se contenter de placer lui-même ses sommets et arcs selon ses besoins.

L'architecture de notre système est assez ouverte pour permettre de lui ajouter de telles fonctionnalités : de nombreux algorithmes de repositionnement d'éléments existent, et ils peuvent être appliqués aux informations d'habillage de nos graphes.

Les modifications à apporter ne modifient en rien l'architecture du système. Il s'agit de nouvelles fonctionnalités à ajouter au système ; certains algorithmes pourront même être certainement construits en se basant sur notre classe abstraite « algorithme ».

8.2.2 Ajouts d'informations aux éléments du graphe

Notre système étant à vocation didactique, il affiche actuellement les graphes sous une forme très simple : des sommets représentés sous des formes géométriques simples (cercles, carrés), des arcs droits entre ces sommets.

Il est important de pouvoir améliorer les types de représentations, mais pas uniquement au point de vue graphique : un sommet du graphe peut représenter, par exemple, une classe dans un diagramme UML, une machine dans un diagramme de réseau informatique, un état complexe dans une machine à état, etc.

On peut ajouter de nouvelles informations aux éléments de base et aux informations d'habillage de ces éléments. Les techniques d'héritage permettront certainement d'offrir de nouvelles possibilités à l'utilisation de notre système.

8.2.3 Emplois de standards pour l'échange de graphes

La plupart des produits découverts utilisent leur propre modèle de description textuelle de graphes, et notre système ne fait pas exception. Il nous semble utile de fournir la possibilité de lire et/ou écrire des graphes décrits dans d'autres langages de description, et notamment un standard comme GDL.

Ici également, l'architecture est suffisamment ouverte pour permettre de développer facilement de telles fonctionnalités.

8.2.4 Génération automatique de graphes

Outre la création interactive de petits graphes, il peut être très utile de disposer de fonctionnalités permettant de générer divers types de graphes très rapidement. Par exemple, demander au système de générer un graphe complet comportant X sommets, avec des poids d'arcs aléatoires pris dans des limités fixées.

Cette fonctionnalité nous semble relever d'un nouvel objet générateur de graphes, qui utiliserait les méthodes de notre classe Graphe actuelle ou améliorée.

9

Bibliographie

Le lecteur trouvera ci-après les références aux ouvrages, travaux et sites *internet* que nous avons consultés dans le cadre de notre travail.

[Buttrick R, 2002]

Buttrick Robert, *Gestion de projet en action*, Editions Village Mondial, Paris, 2002.

[Jacobson I, et al, 2000]

Jacobson Ivar, Booch Grady & Rumbaugh James, *Le processus unifé de développement logiciel*, Eyrolles, Paris, 2000.

[Muller PA, et al, 2000]

Muller Pierre-Alain & Gaertner Nathalie, *Modélisation objet avec UML*, Eyrolles, Paris, 2000.

[Leclercq JP, 1998]

Leclercq Jean-Paul, *Notes (syllabus) du cours de Théorie des Graphes*, 1998.

[GRIN, 2003]

Pechenkin Vitaly, *GRaph INterface (GRIN) software*, http://www.geocities.com/pechv_ru/, Created 10 Sept. 1998, Last modified 22 May 2003, Accessed 12 Feb. 2003.

[AISEE, 2003]

Absint GmbH, *aiSee graph layout software*, http://www.aisee.com, Last modified 7 May 2003, Accessed 15 May 2003.

[JDSL, 2003]

JDSL team, *JDSL: data structures library in Java*, http://www.jdsl.org, Last modified 19 April 2003, Accessed 15 May 2003.

[DGEA, 2003]

Linyuan Lu, *DGEA : Directed graph editor applet*, http://www.math.ucsd.edu/~llu/dgea/, Last modified (unknown), Accessed 15 May 2003.

[JVIEWS, 2003]

ILOG Inc., *Jviews component suite*, http://www.ilog.com/products/jviews, Last modified (unknown), Accessed 15 May 2003.

[VGJ, 2003]

Auburn University Computer Science and Software Engineering department, *Visualizing Graphs with Java*, http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html, Last modified (unknown), Accessed 15 May 2003.

[OpenJGraph, 2002]

Salvo Jesus M, *OpenJGraph – Java graph and graph drawing project*, http://openjgraph.sourceforge.net/, Last modified 5 Oct. 2002, Accessed 15 May 2003.